Chainer Documentation

Release 7.0.0b4

Preferred Networks, inc. and Preferred Infrastructure, inc.

Oct 07, 2019

TUTORIALS

Chainer is a powerful, flexible and intuitive deep learning framework.

- Chainer supports CUDA computation. It only requires a few lines of code to leverage a GPU. It also runs on multiple GPUs with little effort.
- Chainer supports various network architectures including feed-forward nets, convnets, recurrent nets and recursive nets. It also supports per-batch architectures.
- Forward computation can include any control flow statements of Python without lacking the ability of backpropagation. It makes code intuitive and easy to debug.

CHAPTER

ONE

CHAINER AT A GLANCE

Welcome to Chainer!

Chainer is a rapidly growing neural network platform. The strengths of Chainer are:

- Python-based – Chainer is developed in Python, allowing for inspection and customization of all code in python and understandable python messages at run time
- Define by Run – neural networks definitions are defined on-the-fly at run time, allowing for dynamic network changes
- NumPy based syntax for working with arrays, thanks to CuPy implementation
- Fully customizable – since Chainer is pure python, all classes and methods can be adapted to allow for the latest cutting edge or specialized approaches
- Broad and deep support – Chainer is actively used for most of the current approaches for neural nets (CNN, RNN, RL, etc.), aggressively adds new approaches as they're developed, and provides support for many kinds of hardware as well as parallelization for multiple GPUs

1.1 Mushrooms – tasty or deadly?

Let's take a look at a basic program of Chainer to see how it works. For a dataset, we'll work with Kaggle's edible vs. poisonous mushroom dataset, which has over 8,000 examples of mushrooms, labelled by 22 categories including odor, cap color, habitat, etc., in a mushrooms.csv file.

How will Chainer learn which mushrooms are edible and which mushrooms will kill you? Let's see!

The code below is from the glance example in the examples/glance directory.

1.2 Code Breakdown

1.2.1 Initialization

Let's start the program. Here are the typical imports for a Chainer program. *chainer.links* contain trainable parameters and *chainer.functions* do not.

```
import chainer as ch
from chainer import datasets
import chainer.functions as F
import chainer.links as L
from chainer import training
```

(continues on next page)

(continued from previous page)

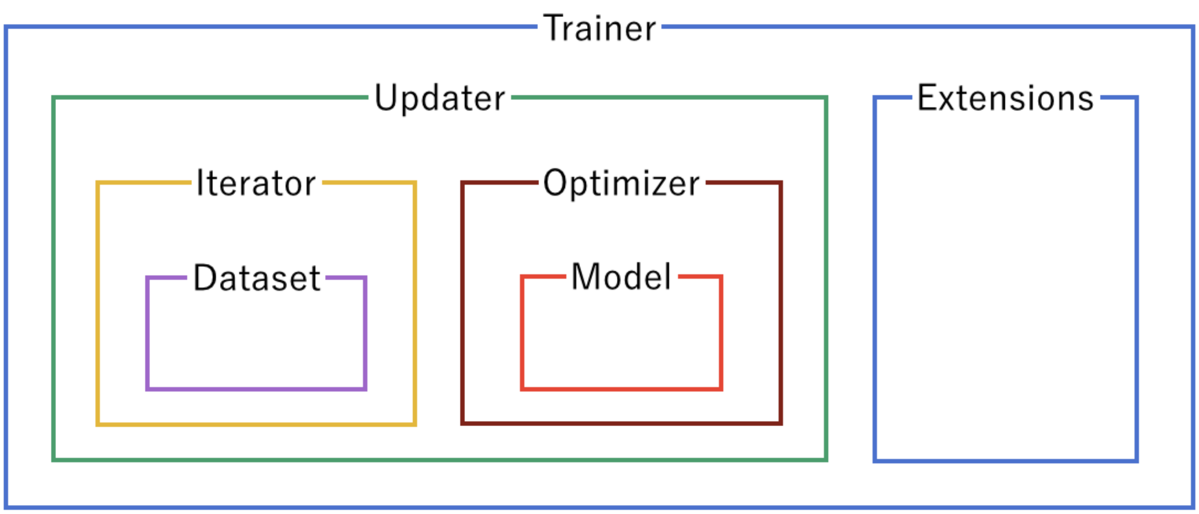
```
from chainer.training import extensions

import numpy as np
```

We'll use Matplotlib for the graphs to show training progress.

```
import matplotlib
matplotlib.use('Agg')
```

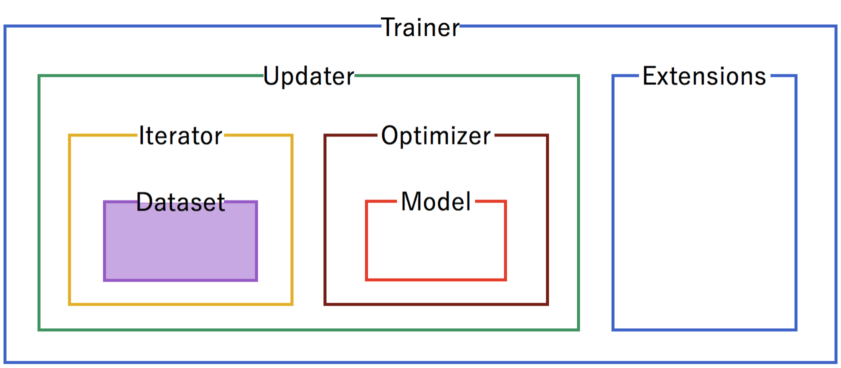
1.2.2 Trainer Structure

A `trainer` is used to set up our neural network and data for training. The components of the `trainer` are generally hierarchical, and are organized as follows:



Each of the components is fed information from the components within it. Setting up the trainer starts at the inner components, and moves outward, with the exception of *extensions*, which are added after the `trainer` is defined.

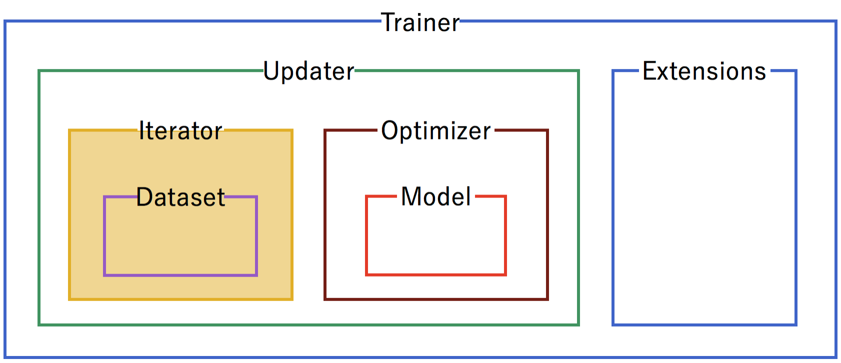
1.2.3 Dataset



Our first step is to format the `dataset`. From the raw mushrooms.csv, we format the data into a Chainer `TupleDataset`.

```
mushroomsfile = 'mushrooms.csv'
data_array = np.genfromtxt(
    mushroomsfile, delimiter=',', dtype=str, skip_header=1)
for col in range(data_array.shape[1]):
    data_array[:, col] = np.unique(data_array[:, col], return_inverse=True)[1]

X = data_array[:, 1:].astype(np.float32)
Y = data_array[:, 0].astype(np.int32)[:, None]
train, test = datasets.split_dataset_random(
    datasets.TupleDataset(X, Y), int(data_array.shape[0] * .7))
```

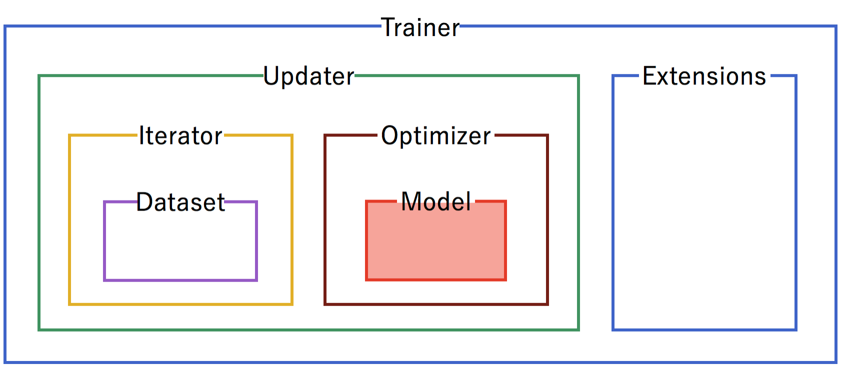
1.2.4 Iterator



Configure `iterators` to step through batches of the data for training and for testing validation. In this case, we'll use a batch size of 100. For the training iterator, repeating and shuffling are implicitly enabled, while they are explicitly disabled for the testing iterator.

```
train_iter = ch.iterators.SerialIterator(train, 100)
test_iter = ch.iterators.SerialIterator(
    test, 100, repeat=False, shuffle=False)
```

1.2.5 Model



Next, we need to define the neural network for inclusion in our model. For our mushrooms, we'll chain together two fully-connected, `Linear`, hidden layers between the input and output layers.

As an activation function, we'll use standard Rectified Linear Units (`relu()`).

Using `Sequential` allows us to define the neural network model in a compact format.

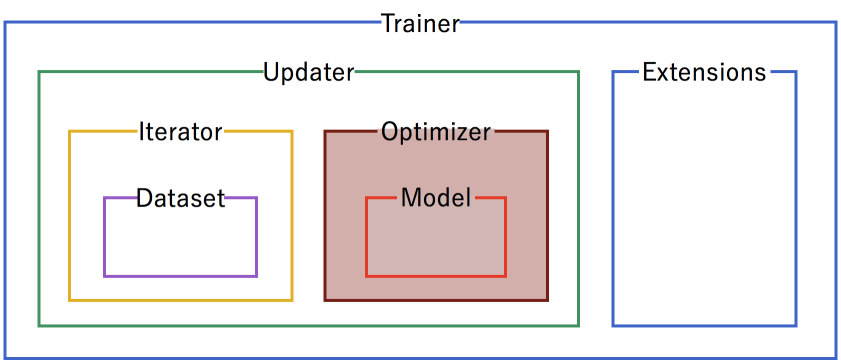
```python
# Network definition
def MLP(n_units, n_out):
    layer = ch.Sequential(L.Linear(n_units), F.relu)
    model = layer.repeat(2)
    model.append(L.Linear(n_out))

    return model
```

Since mushrooms are either edible or poisonous (no information on psychedelic effects!) in the dataset, we'll use a Link *Classifier* for the output, with 44 units (double the features of the data) in the hidden layers and a single edible/poisonous category for classification.

```python
model = L.Classifier(
    MLP(44, 1), lossfun=F.sigmoid_cross_entropy, accfun=F.binary_accuracy)
```

Note that in the two code snippets above we have not specified the size of the input layer. Once we start feeding the neural network with samples, Chainer will recognize the dimensionality of the input automatically and initialize the matrix for each layer with the appropriate shape. In the example above, that is 44×22 for the first hidden layer, 44×44 for the second hidden layer, and 1×44 for the output layer.

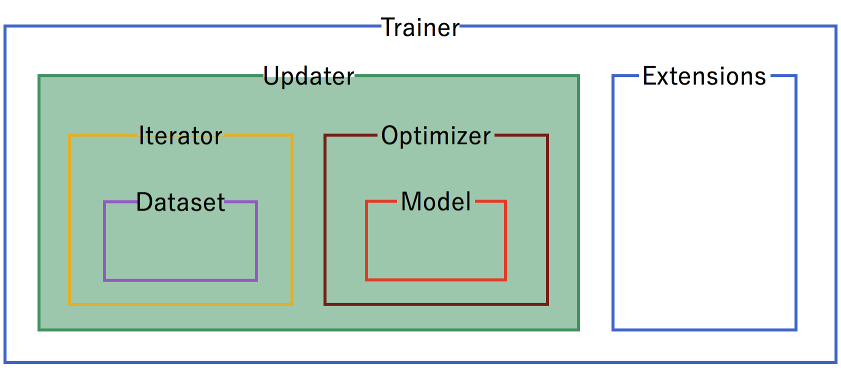
1.2.6 Optimizer



Pick an *optimizer*, and set up the `model` to use it.

```python
# Setup an optimizer
optimizer = ch.optimizers.SGD().setup(model)
```

1.2.7 Updater



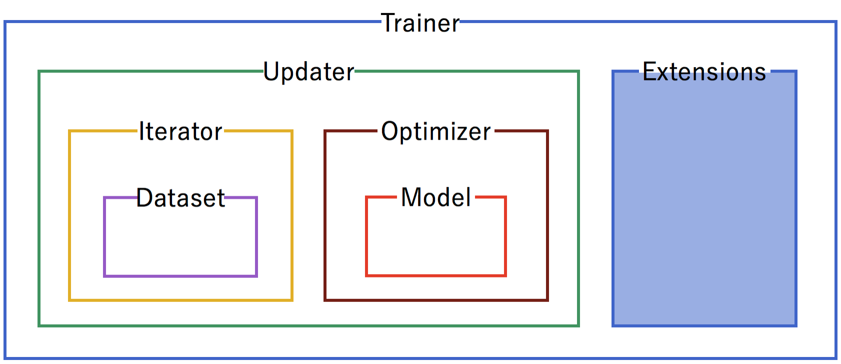
Now that we have the training *iterator* and *optimizer* set up, we link them both together into the *updater*. The *updater* uses the minibatches from the *iterator*, does the forward and backward processing of the model, and updates the parameters of the model according to the *optimizer*. Setting the `device=-1` sets the device as the CPU. To use a GPU, set `device` equal to the number of the GPU, usually `device=0`.

```
# Create the updater, using the optimizer
updater = training.StandardUpdater(train_iter, optimizer, device=-1)
```

Finally we create a *Trainer* object. The `trainer` processes minibatches using the `updater` defined above until a certain stop condition is met and allows the use of extensions during the training. We set it to run for 50 epochs and store all files created by the extensions (see below) in the `result` directory.

```
# Set up a trainer
trainer = training.Trainer(updater, (50, 'epoch'), out='result')
```

1.2.8 Extensions



Extensions can be used to execute code at certain events during the training, such as every epoch or every 1000 iterations. This mechanism is used in Chainer to evaluate models during training, print progress messages, or dump intermediate model files.

First, use the testing *iterator* defined above for an *Evaluator* extension to the trainer to provide test scores. If using a GPU instead of the CPU, set `device` to the ID of the GPU, usually `0`.

```
# Evaluate the model with the test dataset for each epoch
trainer.extend(extensions.Evaluator(test_iter, model, device=-1))
```

Save a computational graph from `loss` variable at the first iteration. `main` refers to the target link of the `main` *optimizer*. The graph is saved in the Graphviz's dot format. The output location (directory) to save the graph is set by the `out` argument of *trainer*.

```
# Dump a computational graph from 'loss' variable at the first iteration
# The "main" refers to the target link of the "main" optimizer.
trainer.extend(extensions.DumpGraph('main/loss'))
```

Take a snapshot of the *trainer* object every 20 epochs.

```
trainer.extend(extensions.snapshot(), trigger=(20, 'epoch'))
```

Write a log of evaluation statistics for each epoch.

```
# Write a log of evaluation statistics for each epoch
trainer.extend(extensions.LogReport())
```

Save two plot images to the result directory.

```
# Save two plot images to the result dir
trainer.extend(
    extensions.PlotReport(['main/loss', 'validation/main/loss'],
                          'epoch', file_name='loss.png'))
trainer.extend(
    extensions.PlotReport(
        ['main/accuracy', 'validation/main/accuracy'],
        'epoch', file_name='accuracy.png'))
```

Print selected entries of the log to standard output.

```
# Print selected entries of the log to stdout
trainer.extend(extensions.PrintReport(
    ['epoch', 'main/loss', 'validation/main/loss',
     'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))
```

1.2.9 Main Loop

Finally, with the `trainer` and all the extensions set up, we can add the line that actually starts the main loop:

```
#  Run the training
trainer.run()
```

1.2.10 Inference

Once the training is complete, only the model is necessary to make predictions. Let's check that a random line from the test data set and see if the inference is correct:

```
x, t = test[np.random.randint(len(test))]

predict = model.predictor(x[None]).array
predict = predict[0][0]

if predict >= 0:
    print('Predicted Poisonous, Actual ' + ['Edible', 'Poisonous'][t[0]])
else:
    print('Predicted Edible, Actual ' + ['Edible', 'Poisonous'][t[0]])
```

1.3 Output

Output for this instance will look like:

```
epoch       main/loss   validation/main/loss  main/accuracy  validation/main/accuracy␣
↪ elapsed_time
1           0.550724    0.502818              0.733509       0.752821                 ␣
↪ 0.215426
2           0.454206    0.446234              0.805439       0.786926                 ␣
↪ 0.902108
3           0.402783    0.395893              0.838421       0.835979                 ␣
↪ 1.50414
```

(continues on next page)

(continued from previous page)

```
4           0.362979    0.359988              0.862807       0.852632              ␣
↪ 2.24171
5           0.32713     0.329881              0.88           0.874232              ␣
↪ 2.83247
6           0.303469    0.31104               0.892456       0.887284              ␣
↪ 3.45173
7           0.284755    0.288553              0.901754       0.903284              ␣
↪ 3.9877
8           0.26801     0.272033              0.9125         0.907137              ␣
↪ 4.54794
9           0.25669     0.261355              0.920175       0.917937              ␣
↪ 5.21672
10          0.241789    0.251821              0.927193       0.917937              ␣
↪ 5.79541
11          0.232291    0.238022              0.93           0.925389              ␣
↪ 6.3055
12          0.222805    0.22895               0.934035       0.923389              ␣
↪ 6.87083
13          0.21276     0.219291              0.93614        0.928189              ␣
↪ 7.54113
14          0.204822    0.220736              0.938596       0.922589              ␣
↪ 8.12495
15          0.197671    0.207017              0.938393       0.936042              ␣
↪ 8.69219
16          0.190285    0.199129              0.941053       0.934842              ␣
↪ 9.24302
17          0.182827    0.193303              0.944386       0.942695              ␣
↪ 9.80991
18          0.176776    0.194284              0.94614        0.934042              ␣
↪ 10.3603
19          0.16964     0.177684              0.945789       0.945242              ␣
↪ 10.8531
20          0.164831    0.171988              0.949825       0.947347              ␣
↪ 11.3876
21          0.158394    0.167459              0.952982       0.949747              ␣
↪ 11.9866
22          0.153353    0.161774              0.956964       0.949347              ␣
↪ 12.6433
23          0.148209    0.156644              0.957368       0.951747              ␣
↪ 13.3825
24          0.144814    0.15322               0.957018       0.955495              ␣
↪ 13.962
25          0.138782    0.148277              0.958947       0.954147              ␣
↪ 14.6
26          0.135333    0.145225              0.961228       0.956695              ␣
↪ 15.2284
27          0.129593    0.141141              0.964561       0.958295              ␣
↪ 15.7413
28          0.128265    0.136866              0.962632       0.960547              ␣
↪ 16.2711
29          0.123848    0.133444              0.966071       0.961347              ␣
↪ 16.7772
30          0.119687    0.129579              0.967193       0.964547              ␣
↪ 17.3311
31          0.115857    0.126606              0.968596       0.966547              ␣
↪ 17.8252
32          0.113911    0.124272              0.968772       0.962547              ␣
↪ 18.3121
```

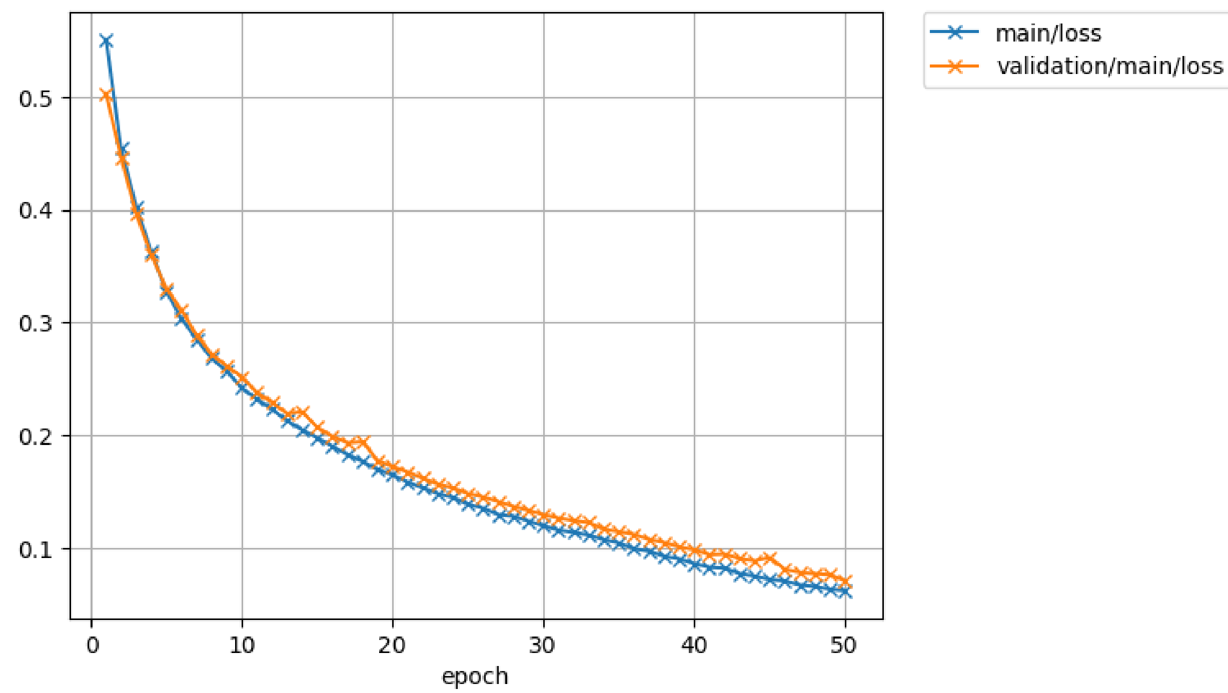
(continues on next page)

(continued from previous page)

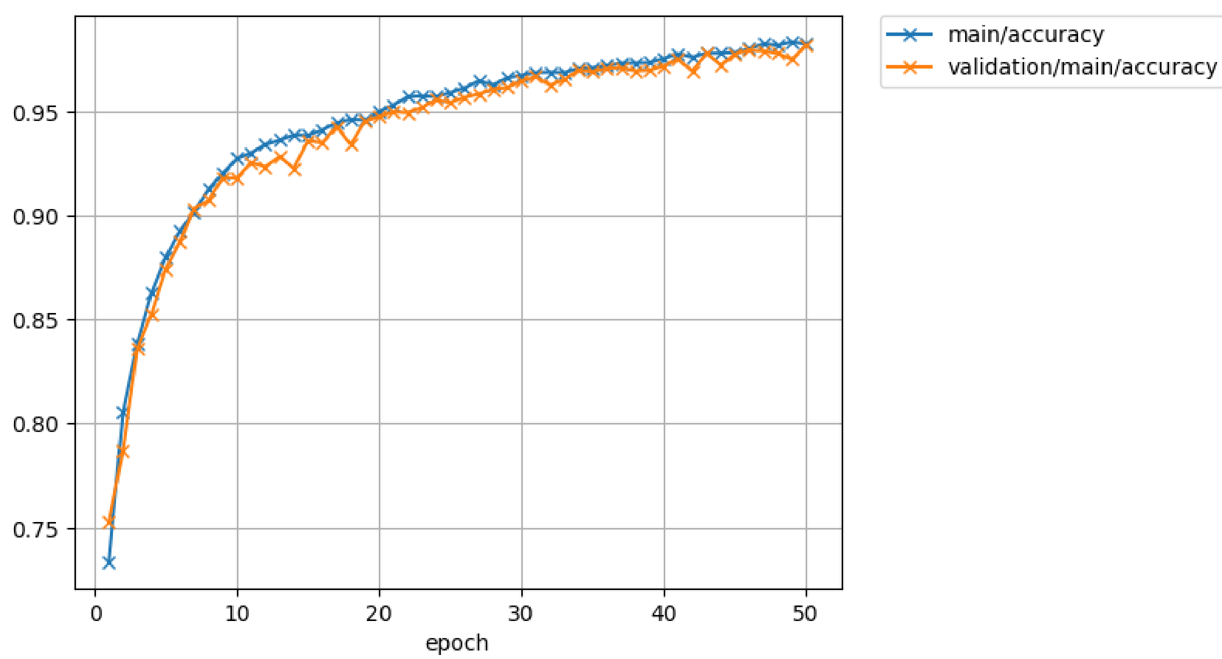
```
33          0.111502    0.122548              0.968596       0.965095                ␣
↪ 18.8973
34          0.107427    0.116724              0.970526       0.969747                ␣
↪ 19.4723
35          0.104536    0.114517              0.970877       0.969095                ␣
↪ 20.0804
36          0.099408    0.112128              0.971786       0.970547                ␣
↪ 20.6509
37          0.0972982   0.107618              0.973158       0.970947                ␣
↪ 21.2467
38          0.0927064   0.104918              0.973158       0.969347                ␣
↪ 21.7978
39          0.0904702   0.101141              0.973333       0.969747                ␣
↪ 22.3328
40          0.0860733   0.0984015             0.975263       0.971747                ␣
↪ 22.8447
41          0.0829282   0.0942095             0.977544       0.974947                ␣
↪ 23.5113
42          0.082219    0.0947418             0.975965       0.969347                ␣
↪ 24.0427
43          0.0773362   0.0906804             0.977857       0.977747                ␣
↪ 24.5252
44          0.0751769   0.0886449             0.977895       0.972147                ␣
↪ 25.1722
45          0.072056    0.0916797             0.978246       0.977495                ␣
↪ 26.0778
46          0.0708111   0.0811359             0.98           0.979347                ␣
↪ 26.6648
47          0.0671919   0.0783265             0.982456       0.978947                ␣
↪ 27.2929
48          0.0658817   0.0772342             0.981754       0.977747                ␣
↪ 27.8119
49          0.0634615   0.0762576             0.983333       0.974947                ␣
↪ 28.3876
50          0.0622394   0.0710278             0.982321       0.981747                ␣
↪ 28.9067
Predicted Edible Actual Edible
```

Our prediction was correct. Success!

The loss function:



And the accuracy



CHAPTER

TWO

CONCEPTS WALKTHROUGH

2.1 Define-by-Run

As mentioned on the top page, Chainer is a flexible framework for neural networks. One major goal is flexibility, so it must enable us to write complex architectures simply and intuitively.

Most existing deep learning frameworks are based on the **"Define-and-Run"** scheme. That is, first a network is defined and fixed, and then the user periodically feeds it with mini-batches of training data. Since the network is statically defined before any forward/backward computation, all the logic must be embedded into the network architecture as *data*. Consequently, defining a network architecture in such systems (e.g. Caffe) follows a declarative approach. Note that one can still produce such a static network definition using imperative languages (e.g. torch.nn, Theano-based frameworks, and TensorFlow).

In contrast, Chainer adopts a **"Define-by-Run"** scheme, i.e., the network is defined dynamically via the actual forward computation. More precisely, Chainer stores the history of computation instead of programming logic. This strategy enables us to fully leverage the power of programming logic in Python. For example, Chainer does not need any magic to introduce conditionals and loops into the network definitions. The Define-by-Run scheme is the core concept of Chainer. We will show in this tutorial how to define networks dynamically.

This strategy also makes it easy to write multi-GPU parallelization, since logic comes closer to network manipulation. We will review such amenities in later sections of this tutorial.

2.2 Variables and Derivatives

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

As described previously, Chainer uses the “Define-by-Run” scheme, so forward computation itself *defines* the network. In order to start forward computation, we have to set the input array to a `chainer.Variable` object. Here we start with a simple `ndarray` with only one element:

```
>>> x_data = np.array([5], dtype=np.float32)
>>> x = Variable(x_data)
```

A Variable object supports basic arithmetic operators. In order to compute $y = x^2 - 2x + 1$, just write:

```
>>> y = x**2 - 2 * x + 1
```

The resulting `y` is also a Variable object, whose value can be extracted by accessing the `array` attribute:

```
>>> y.array
array([16.], dtype=float32)
```

Note: `Variable` has two attributes to represent the underlying array: `array` and `data`. There is no difference between the two; both refer to exactly the same object. However it is not recommended that you use `.data` because it might be confused with `numpy.ndarray.data` attribute.

What `y` holds is not only the result value. It also holds the history of computation (or computational graph), which enables us to compute its derivative. This is done by calling its `backward()` method:

```
>>> y.backward()
```

This runs *error backpropagation* (a.k.a. *backprop* or *reverse-mode automatic differentiation*). Then, the gradient is computed and stored in the `grad` attribute of the input variable `x`:

```
>>> x.grad
array([8.], dtype=float32)
```

Also we can compute gradients of intermediate variables. Note that Chainer, by default, releases the gradient arrays of intermediate variables for memory efficiency. In order to preserve gradient information, pass the `retain_grad` argument to the backward method:

```
>>> z = 2*x
>>> y = x**2 - z + 1
>>> y.backward(retain_grad=True)
>>> z.grad
array([-1.], dtype=float32)
```

All these computations can be generalized to a multi-element array input. While single-element arrays are automatically initialized to `[1]`, to start backward computation from a variable holding a multi-element array, we must set the *initial error* manually. This is done simply by setting the `grad` attribute of the output variable:

```
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = x**2 - 2*x + 1
>>> y.grad = np.ones((2, 3), dtype=np.float32)
>>> y.backward()
>>> x.grad
array([[ 0.,  2.,  4.],
       [ 6.,  8., 10.]], dtype=float32)
```

Note: Many functions taking `Variable` object(s) are defined in the `chainer.functions` module. You can combine them to realize complicated functions with automatic backward computation.

Note: Instead of using *backward()*, you can also calculate gradients of any variables in a computational graph w.r.t. any other variables in the graph using the *chainer.grad()* function.

2.2.1 Higher-Order Derivatives

Variable also supports higher-order derivatives (a.k.a. double backpropagation).

Let's see a simple example. First calculate the first-order derivative. Note that `enable_double_backprop=True` is passed to `y.backward()`.

```
>>> x = chainer.Variable(np.array([[0, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = x ** 3
>>> y.grad = np.ones((2, 3), dtype=np.float32)
>>> y.backward(enable_double_backprop=True)
>>> x.grad_var
variable([[  0.,  12.,  27.],
          [ 48.,  75., 108.]])
>>> assert x.grad_var.array is x.grad
>>> assert (x.grad == (3 * x**2).array).all()
```

chainer.Variable.grad_var is a *Variable* for *chainer.Variable.grad* (which is an `ndarray`). By passing `enable_double_backprop=True` to `backward()`, a computational graph for the backward calculation is recorded. So, you can start backpropagation from `x.grad_var` to calculate the second-order derivative.

```
>>> gx = x.grad_var
>>> x.cleargrad()
>>> gx.grad = np.ones((2, 3), dtype=np.float32)
>>> gx.backward()
>>> x.grad
array([[ 0., 12., 18.],
       [24., 30., 36.]], dtype=float32)
>>> assert (x.grad == (6 * x).array).all()
```

2.3 Links

In order to write neural networks, we have to combine functions with *parameters* and optimize the parameters. You can use the class *Link* to do this. A *Link* is an object that holds parameters (i.e. optimization targets).

The most fundamental ones are links that behave like regular functions while replacing some arguments by their parameters. We will introduce higher level links, but here think of links as simply functions with parameters.

One of the most frequently used links is the *Linear* link (a.k.a. *fully-connected layer* or *affine transformation*). It represents a mathematical function $f(x) = Wx + b$, where the matrix W and the vector b are parameters. This link corresponds to its pure counterpart *linear()*, which accepts x, W, b as arguments. A linear link from three-dimensional space to two-dimensional space is defined by the following line:

```
>>> f = L.Linear(3, 2)
```

Note: Most functions and links only accept mini-batch input, where the first dimension of the input array is considered as the *batch dimension*. In the above Linear link case, input must have shape of $(N, 3)$, where N is the mini-batch size.

The parameters of a link are stored as attributes. Each parameter is an instance of `Variable`. In the case of the Linear link, two parameters, `W` and `b`, are stored. By default, the matrix `W` is initialized randomly, while the vector `b` is initialized with zeros. This is the preferred way to initialize these parameters.

```
>>> f.W.array
array([[ 1.0184761 ,  0.23103087,  0.5650746 ],
       [ 1.2937803 ,  1.0782351 , -0.56423163]], dtype=float32)
>>> f.b.array
array([0., 0.], dtype=float32)
```

An instance of the Linear link acts like a usual function:

```
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = f(x)
>>> y.array
array([[3.1757617, 1.7575557],
       [8.619507 , 7.1809077]], dtype=float32)
```

Note: Sometimes it is cumbersome to compute the dimension of the input space. The linear link and some of (de)convolution links can omit the input dimension in their instantiation and infer it from the first mini-batch.

For example, the following line creates a linear link whose output dimension is two:

```
>>> f = L.Linear(2)
```

If we feed a mini-batch of shape $(2, M)$, the input dimension will be inferred as `M`, which means `l.W` will be a 2 x M matrix. Note that its parameters are initialized in a lazy manner at the first mini-batch. Therefore, `l` does not have `W` attribute if no data is put to the link.

Gradients of parameters are computed by the `backward()` method. Note that gradients are **accumulated** by the method rather than overwritten. So first you must clear the gradients to renew the computation. It can be done by calling the `cleargrads()` method.

```
>>> f.cleargrads()
```

Now we can compute the gradients of parameters by simply calling the backward method and access them via the `grad` property.

```
>>> y.grad = np.ones((2, 2), dtype=np.float32)
>>> y.backward()
>>> f.W.grad
array([[5., 7., 9.],
       [5., 7., 9.]], dtype=float32)
>>> f.b.grad
array([2., 2.], dtype=float32)
```

2.4 Define your own function

In this section, you will learn about the following things:

- How to define a function on variables
- Useful tools to write a function using a GPU
- How to test the function definition

After reading this section, you will be able to:

- Write your own functions
- Define simple kernels in the function definition

  In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

2.4.1 Differentiable Functions

Chainer provides a collection of functions in the *chainer.functions* module. It covers typical use cases in deep learning, so many existing works can be implemented with them. On the other hand, deep learning is evolving rapidly and we cannot cover all possible functions to define unseen architectures. So it is important to learn how to define your own functions.

2.4.2 New-Style v.s. Old-Style Functions

In Chainer, you can define a function in two ways: new-style and old-style.

- New-style functions inherit from *chainer.FunctionNode* class (introduced in Chainer v3). Forward computation can be implemented using NumPy/CuPy. Backward computation needs to be implemented by using (possibly a composition of) other new-style functions.
- Old-style functions inherit from *chainer.Function* class. Forward and backward computation can be implemented using NumPy/CuPy.

The primary advantage of using new-style functions is that they support computation of higher-order gradients (a.k.a. higher-order derivative or double backpropagation). Higher-order gradients are used in some models e.g., recently-proposed GAN architectures. New-style functions are also better in terms of performance of backward, as the interface allows an implementation to skip the computation of unneeded input gradients.

Currently, most of *built-in functions* are implemented in new-style (with a few exceptions listed in #4449). Basically, we recommend you use new-style when implementing new functions. However, you can still continue to use existing old-style functions for the foreseeable future.

In the following sections, we describe steps to implenent user-defiend functions in new-style. You can also refer to *Implementing Old-Style Functions* and *Migrating From Old-Style Functions To New-Style Functions* if you have interest.

2.4.3 Implementing New-Style Functions

First, suppose we want to define an elementwise function $f(x, y, z) = x * y + z$. While it is possible to implement this equation using a combination of the `*` and `+` functions, defining it as a single function may reduce memory consumption, so it is *not* only a toy example. Here we call this function *MulAdd*.

Let's start with defining MulAdd working on the CPU. New-style functions must inherit the `chainer.FunctionNode` class. The skeleton of a function looks like:

```
class MulAdd(FunctionNode):
    def forward_cpu(self, inputs):
        # do forward computation on CPU
        return some_tuple

    def backward(self, target_input_indexes, grad_outputs):
        # do backward computation
        return some_tuple
```

We must implement `forward_cpu()` and `backward()` methods.

- In `forward_cpu()` function, `inputs` is a tuple of array(s). You need to return a tuple of array(s), which is a result of forward computation.
- In `backward()` function, `grad_outputs` is a tuple of `Variable`(s) which are gradients with regard to each output(s), i.e., the length of `grad_outputs` tuple equals to the number of outputs returned by `forward_cpu`). You need to return a tuple of `Variable`(s) which are gradients with regard to each input(s), i.e., the length of returned tuple equals to the number of inputs to `forward_cpu`. You can optionally use `target_input_indexes` (a tuple of indices required to compute gradients) to omit computing unnecessary gradients. We will show you the usage of `target_input_indexes` later.

Warning: Be careful to return a tuple even if you have just one array or Variable to return.

Note: Unlike old-style functions, inputs and outputs of backward method in new-style functions are `Variable`s. In other words, the backward method is device agnostic; there are no `backward_cpu` or `backward_gpu` in `FunctionNode`.

MulAdd is simple and can be implemented as follows:

```
class MulAdd(FunctionNode):
    def forward_cpu(self, inputs):
        # Unpack input arrays (``numpy.ndarray``).
        x, y, z = inputs

        # Mark inputs (``x`` and ``y``) as retained so that it can be
        # accessed during the backward process.
```

(continues on next page)

(continued from previous page)

```
        self.retain_inputs((0, 1))

        # Compute results.
        w = x * y + z

        # Return the result as a tuple.
        return w,

    def backward(self, target_input_indexes, grad_outputs):
        # Unpack inputs retained in the forward process (``Variable``).
        x, y = self.get_retained_inputs()

        # Get gradients w.r.t. the output (Variable).
        gw, = grad_outputs

        # Compute gradients w.r.t the inputs.
        gx = y * gw
        gy = x * gw
        gz = gw

        # Return the result as a tuple.
        return gx, gy, gz
```

As per the warning above, the *forward_cpu()* method returns a tuple of single element. Note that all arrays appearing in `forward_cpu` are `numpy.ndarray`. The forward function is straightforward; it unpacks the input tuple, computes the output, and packs it into a tuple. The backward function is a bit more complicated. Recall the rule of differentiation of multiplication. This example just implements the rule. Look at the return values, the function just packs the gradient of each input in the same order and returns them.

By just defining the core computation of forward and backward, *FunctionNode* class provides a chaining logic on it (i.e., storing the history of computation, etc.).

Note: Assuming we implement a (forward) function $y = f(x)$ which takes as input the vector $x \in \mathbb{R}^n$ and produces as output a vector $y \in \mathbb{R}^m$. Then the `backward` method has to compute

$$\lambda_i = \sum_{j=1}^{m} \frac{\partial y_j}{\partial x_i} \gamma_j \text{ for } i = 1 \dots n$$

where γ is the `grad_outputs`. Note, that the resulting vector λ must have the same shape as the arguments of the `forward` method.

Now let's define the corresponding GPU method. You can easily predict that the method we have to write is named *forward_gpu()*:

```
class MulAdd(FunctionNode):
    def forward_cpu(self, inputs):
        ...

    def forward_gpu(self, inputs):
        # Unpack input arrays (``cupy.ndarray``).
        x, y, z = inputs

        # Mark inputs (``x`` and ``y``) as retained so that it can be
        # accessed during the backward process.
```

(continues on next page)

(continued from previous page)

```
        self.retain_inputs((0, 1))

        # Compute results.
        w = x * y + z

        # Return the result as a tuple.
        return w,

    def backward(self, target_input_indexes, grad_outputs):
        ...
```

In `forward_gpu` method, arrays are of type `cupy.ndarray`. We use arithmetic operators defined for this class. These operators implement the basic elementwise arithmetics.

You may find that the definitions of `forward_gpu` is exactly same as `forward_cpu`. In that case, we can reduce them io `forward()`.

```
class MulAdd(FunctionNode):
    def forward(self, inputs):
        # Unpack input arrays (``numpy.ndarray`` or ``cupy.ndarray``).
        x, y, z = inputs

        # Mark inputs (``x`` and ``y``) as retained so that it can be
        # accessed during the backward process.
        self.retain_inputs((0, 1))

        # Compute results.
        w = x * y + z

        # Return the result as a tuple.
        return w,

    def backward(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

Since the `cupy.ndarray` class implements many methods of `numpy.ndarray`, we can write these unified methods in most cases.

The MulAdd function can be used as follows:

```
x = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
y = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
z = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
w, = MulAdd().apply((x, y, z))
```

It looks a bit ugly: we have to explicitly instantiate MulAdd before applying it to variables. We also have to be careful that one instance of MulAdd must not be used multiple times, since it acts as a node in the computational graph. In Chainer, we often define a thin wrapper Python function that hide the instantiation:

```
def muladd(x, y, z):
    return MulAdd().apply((x, y, z))
```

(continues on next page)

(continued from previous page)

```
w = muladd(x, y, z)
```

All functions under `chainer.functions` are implemented as wrapper functions like this.

Unified forward/backward methods with NumPy/CuPy functions

CuPy implements many functions that are compatible to those of NumPy. We can write unified forward/backward methods with them. Consider that we want to write a backprop-able function $f(x, y) = \exp(x) + \exp(y)$. We name it *ExpAdd* here. It can be written straight-forward as follows:

```
from chainer.backends import cuda

class ExpAdd(FunctionNode):
    def forward_cpu(self, inputs):
        self.retain_inputs((0, 1))
        x, y = inputs
        z = np.exp(x) + np.exp(y)
        return z,

    def forward_gpu(self, inputs):
        self.retain_inputs((0, 1))
        cupy = cuda.cupy
        x, y = inputs
        z = cupy.exp(x) + cupy.exp(y)
        return z,

    def backward(self, target_input_indexes, grad_outputs):
        x, y = self.get_retained_inputs()
        gz, = grad_outputs

        gx = gz * F.exp(x)
        gy = gz * F.exp(y)
        return gx, gy

def expadd(x, y):
    z, = ExpAdd().apply((x, y))
    return z
```

Note: Here we used `chainer.backends.cuda.cupy` instead of directly accessing `cupy`. This is because the `cupy` module cannot be imported if the CUDA is not installed. In order to keep the implementation valid in non-CUDA environment, we have to defer the access to the `cupy` module. Note that the `chainer.backends.cuda` module can be imported even if the CUDA is not installed. Of course, the module in such environment is almost useless, but if the interpreter does not run through the code accessing CUDA-dedicated functions, the code is still valid.

The CPU and GPU implementations are almost same, except that `numpy` is replaced by `cupy` in `forward_gpu`. We can unify these functions using the `chainer.backend.get_array_module()` function. This function accepts arbitrary number of arrays, and returns an appropriate module for them. See the following code:

```
class ExpAdd(FunctionNode):
    def forward(self, inputs):
        self.retain_inputs((0, 1))
```

(continues on next page)

(continued from previous page)

```
        xp = backend.get_array_module(*inputs)
        x, y = inputs
        z = xp.exp(x) + xp.exp(y)
        return z,

    def backward(self, target_input_indexes, grad_outputs):
        x, y = self.get_retained_inputs()
        gz, = grad_outputs

        gx = gz * F.exp(x)
        gy = gz * F.exp(y)
        return gx, gy

def expadd(x, y):
    z, = ExpAdd().apply((x, y))
    return z
```

Note that this code works correctly even if CUDA is not installed in the environment. If CUDA is not found, *get_array_module()* function always returns `numpy`. We often use the name `xp` for the variadic module name, which is analogous to the abbreviation `np` for NumPy and `cp` for CuPy.

Write an Elementwise Kernel Function

Let's turn back to the MulAdd example.

The GPU implementation of MulAdd as shown above is already fast and parallelized on GPU cores. However, it invokes two kernels during each of forward (`w = x * y + z`) and backward (`gx = y * gw` and `gy = x * gw`) computations. It might hurt performance, since the intermediate temporary arrays are read and written by possibly different GPU cores, which consumes much bandwidth. We can reduce the number of invocations by defining our own kernel. It also reduce the memory consumption.

CuPy provides a useful tool to define elementwise kernels, the `cupy.ElementwiseKernel` class, and Chainer wraps it by *chainer.backends.cuda.elementwise()* function. Our MulAdd implementation can be improved as follows:

```
class MulAdd(FunctionNode):
    def forward_cpu(self, inputs):
        self.retain_inputs((0, 1))
        x, y, z = inputs
        w = x * y + z
        return w,

    def forward_gpu(self, inputs):
        self.retain_inputs((0, 1))
        x, y, z = inputs
        w = cuda.cupy.elementwise(
            'float32 x, float32 y, float32 z',
            'float32 w',
            'w = x * y + z',
            'muladd_fwd')(x, y, z)
        return w,

    def backward(self, target_input_indexes, grad_outputs):
        x, y, z = self.get_retained_inputs()
        gw, = grad_outputs
```

(continues on next page)

(continued from previous page)

```
        return MulAddGrad().apply((x, y, z, gw))

class MulAddGrad(FunctionNode):
    def forward_cpu(self, inputs):
        x, y, z, gw = inputs
        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz

    def forward_gpu(self, inputs):
        x, y, z, gw = inputs
        gx, gy = cuda.elementwise(
            'float32 x, float32 y, float32 gw',
            'float32 gx, float32 gy',
            '''
               gx = y * gw;
               gy = x * gw;
            ''',
            'muladd_bwd')(x, y, gw)

        gz = gw
        return gx, gy, gz

    def backward(self, target_input_indexes, grad_outputs):
        # You can leave this unimplemented unless you need to compute
        # higher-order derivative using this function.
        raise NotImplementedError()
```

`chainer.backends.cuda.elementwise()` function accepts the essential implementation of the kernel function, and returns a kernel invocation function (actually, it returns `ElementwiseKernel` object, which is callable). In typical usage, we pass four arguments to this function as follows:

1. Input argument list. This is a comma-separated string each entry of which consists of a type specification and an argument name.
2. Output argument list in the same format as the input argument list.
3. Body of *parallel loop*. We can use the input/output argument names as an element of these arrays.
4. Name of the kernel function, which is shown in debuggers and profilers.

Above code is not compiled on every forward/backward computation thanks to two caching mechanisms provided by `chainer.backends.cuda.elementwise()`.

The first one is *binary caching*: `chainer.backends.cuda.elementwise()` function caches the compiled binary in the `$(HOME)/.cupy/kernel_cache` directory with a hash value of the CUDA code, and reuses it if the given code matches the hash value. This caching mechanism is actually implemented in CuPy.

The second one is *upload caching*: Given a compiled binary code, we have to upload it to the current GPU in order to execute it. `chainer.backends.cuda.elementwise()` function memoizes the arguments and the current device, and if it is called with the same arguments for the same device, it reuses the previously uploaded kernel code.

The above MulAdd code only works for float32 arrays. The `ElementwiseKernel` also supports the type-variadic kernel definition. In order to define variadic kernel functions, you can use *type placeholder* by placing a single character as type specifier:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
```

(continues on next page)

(continued from previous page)

```
        ...

    def backward_cpu(self, inputs, grad_outputs):
        ...

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y, z = inputs
        w = cuda.elementwise(
            'T x, T y, T z',
            'T w',
            'w = x * y + z',
            'muladd_fwd')(x, y, z)
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx, gy = cuda.elementwise(
            'T x, T y, T gw',
            'T gx, T gy',
            '''
               gx = y * gw;
               gy = x * gw;
            ''',
            'muladd_bwd')(x, y, gw)

        gz = gw
        return gx, gy, gz
```

The type placeholder `T` indicates an arbitrary data type that CuPy supports.

There are more functionalities on user-defined kernels in CuPy. See the CuPy documentation on user-defined kernels for more details.

2.4.4 Advanced Topics

Write a function with training/test mode

We sometimes want to make a function behave differently in training and test modes. The training/test mode in Chainer is configured by *chainer.config*. This is a thread-local configuration object, and users can substitute True or False to its `train` attribute. You can refer to *Configuring Chainer* to see how to configure this flag as well as other configuration items.

Here, we just show how to use this flag to make a function support training/test mode. You will need to check the value of the boolean flag `chainer.config.train` and branch appropriately.

For example, consider the following simple dropout function:

```
def dropout(x):
    xp = backend.get_array_module(x.array)
    mask = 2 * (xp.random.rand(*x.shape) > 0.5).astype(x.dtype)
    return x * mask
```

This function applies dropout to each element and doubles survived elements to preserve the scale. The above implementation applies dropout even in test mode, but it is not a desired behavior. We can fix it as follows:

```
def dropout(x):
    if not chainer.config.train:
        return x

    xp = backend.get_array_module(x.array)
    mask = 2 * (xp.random.rand(*x.shape) > 0.5).astype(x.dtype)
    return x * mask
```

The function now supports test mode. Note that you usually do not have to implement your own dropout function because `dropout()` is officially provided.

Testing Functions

In order to isolate the cause of learning failure from implementation bugs, it is important to test function implementations. Chainer provides simple utilities to help writing unit tests. They are defined in the `gradient_check` module.

The most important test utility is the `numerical_grad()` function. This function computes the numerical gradient of given function using finite differences. It can be used as follows:

```
x  = np.random.randn(4, 3).astype(np.float32)
gy = np.ones((4, 3), dtype=np.float32)
f  = lambda: (x * x,)
gx = gradient_check.numerical_grad(f, (x,), (gy,))
```

`f` is a closure that returns a tuple of array(s) computed from input arrays. The second and third arguments of `numerical_grad()` are tuples of input arrays and output gradient arrays, respectively. The code above computes the numerical gradients of `sum(f(x))`, where `sum` indicates the summation over all elements. The summation can be weighted by changing `gy`. `numerical_grad()` function also accepts additional `eps` argument, which indicates the quantization width of finite differences.

Note: `numerical_grad()` function accepts both CPU and GPU arrays. Note that we cannot mix CPU and GPU arrays.

Another utility is `chainer.testing.assert_allclose()` function. This is similar to `numpy.testing.assert_allclose()` function. The difference is that Chainer's version accepts CPU and GPU arrays as inputs. We can mix them in one invocation of `chainer.testing.assert_allclose()`. The default values of optional arguments are also different.

Here is a typical usage of gradient checking utilities. This is a test example of `functions.relu()` function:

```
import unittest

from chainer import testing

class TestReLU(unittest.TestCase):
    def test_backward_cpu(self):
        x = Variable(np.random.randn(3, 2).astype(np.float32))
        y = F.relu(x)
        y.grad = np.random.randn(3, 2).astype(np.float32)
        y.backward(retain_grad=True)
```

(continues on next page)

(continued from previous page)

```
        def f():
            return F.relu(x).array,

        gx, = gradient_check.numerical_grad(f, (x.array,), (y.grad,))
        testing.assert_allclose(gx, x.grad)
```

The first four lines of the test code are simple forward and backward computation of ReLU function. The next two lines compute numerical gradient using the same forward function without backward routine. And at last, we compare these two results elementwise. Note that the above test code can be easily modified to test GPU version just by replacing CPU arrays to GPU arrays.

In most cases, we do not write the code like the above explicitly because Chainer offers a utility function `chainer.gradient_check.check_backward()` that follows this procedure.

```
import unittest

from chainer import gradient_check

class TestReLU(unittest.TestCase):
    def test_backward_cpu(self):

        def f(x):
            return F.relu(x)

        x = np.random.randn(3, 2).astype(np.float32)
        y_grad = np.random.randn(3, 2).astype(np.float32)

        gradient_check.check_backward(f, x, y_grad, atol=1e-4, rtol=1e-4)
```

You can find many examples of function tests under tests/chainer_tests/functions_tests directory.

You can use `chainer.gradient_check.check_double_backward()` to run gradient check for the second order gradient computed by new-style functions. This function runs two backwpropagations; first to compute the gradient `gx` of `y` w.r.t. `x`, and second to compute the gradient of `gx` w.r.t. `x`. It can be used like `check_backward()`, but `check_double_backward()` expects an additional argument `x_grad_grad`, which is an array or a tuple of arrays used for initializing the gradient array of each gradient w.r.t. an input. In other words, this argument is used to initialize `gx.grad` for the second backprop.

2.4.5 Implementing User-Defined Links

Some functions are meant to be combined with parameters. In such case, it is useful to write a small **link** that wraps the function. We have already seen how to define a chain that wraps other links (by inheriting `Chain` class) in *Creating Models*. Here we study how to define a link that does not hold any other links.

As the first example, suppose that we want to implement elementwise product function between the input array and the parameter array. It can be defined as follows:

```
class EltwiseParamProduct(Link):
    def __init__(self, shape):
        super(EltwiseParamProduct, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(initializers.Normal(scale=1.), shape)

    def __call__(self, x):
        return self.W * x
```

For another example, assume we want to define a simple linear layer. It is already defined as *chainer.links.Linear*, so this is an educational example. The linear layer is divided into two parts: a function and its wrapper link. First, we have to define a function on variables:

```
class LinearFunction(FunctionNode):
    def forward(self, inputs):
        x, W, b = inputs
        return x.dot(W.T) + b,

    def backward(self, inputs, grad_outputs):
        x, W, b = inputs
        gy, = grad_outputs

        gx = gy.dot(W)
        gW = gy.T.dot(x)
        gb = gy.sum(axis=0)
        return gx, gW, gb

def linear(x, W, b):
    return LinearFunction()(x, W, b)
```

This function takes three arguments: input, weight, and bias. It can be used as a part of model definition, though is inconvenient since the user have to manage the weight and bias parameters directly. In order to make a convenient module, let's wrap it into a link:

```
class Linear(Link):
    def __init__(self, in_size, out_size):
        super(Linear, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(
                initializers.Normal(1. / math.sqrt(in_size)),
                (out_size, in_size))
            self.b = chainer.Parameter(0, (out_size,))

    def __call__(self, x):
        return linear(x, self.W, self.b)
```

This link hides the parameters of the linear layer.

Note: An advanced tip to implement functions: if you want to preserve some information between forward and backward computations (e.g. to cache some arrays), you can store it as attributes. Be careful that it might increase the memory consumption during the whole forward-backward computation. If you want to train very large networks on a GPU with limited memory, it is not recommended that you cache arrays between forward and backward. There is one exception for this: caching the output arrays does not change the memory consumption, because they are also held by the output Variable objects.

Warning: You should not assume a one-to-one match of calls of forward and backward. Some users may call backward more than once after one forward call.

2.4.6 Migrating From Old-Style Functions To New-Style Functions

Here are the key differences between *Function* and *FunctionNode*.

- Implementing forward computation (difference between `chainer.Function.forward()` and `chainer.FunctionNode.forward()`)
  - There are no difference between `Function` and `FunctionNode` except that the input arrays are NOT retained by default.

    If you want the inputs to be retained to use them in `backward`, call `retain_inputs()` explicitly. In other words, `self.retain_inputs(())` has no effect in `FunctionNode`.
- Implementing backward computation (difference between `chainer.Function.backward()` and `chainer.FunctionNode.backward()`)
  - Arguments to the method has been changed.
    * `inputs` argument is no longer passed.

      You can use `get_retained_inputs()` and `get_retained_outputs()` to retrieve the inputs/outputs retained in the `forward` method. Note that `grad_outputs` and these retained inputs/outputs are all given as `Variable` objects, and `backward` method must return a tuple of `Variable` objects.
    * `target_input_indexes` argument has been added.

      It contains a sorted indices of the input variables w.r.t. which the gradients are required. You can use it to skip calculation of unneeded gradients. The use of `target_input_indexes` is optional; it is acceptable to calculate and return all gradients.
  - All inputs (`grad_outputs`) and retained values are given in `Variable` in `FunctionNode`, whereas `ndarray` in `Function`.
- Invoking forward computation
  - `Function` is a callable, whereas `FunctionNode` is not.

    You need to use `f.apply((x,))` instead of `f(x)`. Note that `apply()` always returns outputs as `tuple` even if the function generates only one output value.

When migrating from old-style to new-style, typically you will need to write a new function class that implements the first-order gradient of the original function. Here is an example of rewriting old-style `MyOldFunc` unary function to new-style `MyFunc` function.

```
class MyOldFunc(chainer.Function):

    def forward(self, inputs):
        x, = inputs
        ...  # forward computation code
        return y,

    def backward(self, inputs, grad_outputs):
        x, = inputs
        gy, = grad_outputs
        ...  # backward computation code
        return gx,
```

```
class MyFunc(chainer.FunctionNode):

    def forward(self, inputs):
        self.retain_inputs((0,))
        x, = inputs
        ...  # forward computation code in MyOldFunc
        return y,
```

(continues on next page)

(continued from previous page)

```
    def backward(self, target_input_indexes, grad_outputs):
        x, = self.get_retained_inputs()
        gy, = grad_outputs
        gx, = MyFuncGrad().apply((x, gy))
        return gx,

class MyFuncGrad(chainer.FunctionNode):

    def forward(self, inputs):
        x, gy = inputs
        ...  # backward computation code in MyOldFunc
        return gx,

    def backward(self, target_input_indexes, grad_outputs):
        # You can leave this unimplemented unless you need to compute
        # higher-order derivative using this function.
        raise NotImplementedError()
```

2.4.7 Implementing Old-Style Functions

Note: As noted in the *New-Style v.s. Old-Style Functions*, we recommend that you use new-style for newly implemented functions. This section uses the same example as in *Implementing New-Style Functions* but using old-style.

First, suppose we want to define an elementwise function $f(x, y, z) = x * y + z$. While it is possible to implement this equation using a combination of the `*` and `+` functions, defining it as a single function may reduce memory consumption, so it is *not* only a toy example. Here we call this function *MulAdd*.

Let's start with defining MulAdd working on the CPU. Old-style functions must inherit the `Function` class. The skeleton of a function looks like:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        # do forward computation on CPU
        return some_tuple

    def backward_cpu(self, inputs, grad_outputs):
        # do backward computation on CPU
        return some_tuple
```

We must implement `forward_cpu()` and `backward_cpu()` methods. The non-self arguments of these functions are tuples of array(s), and these functions must return a tuple of array(s).

Warning: Be careful to return a tuple of arrays even if you have just one array to return.

MulAdd is simple and implemented as follows:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        x, y, z = inputs
        w = x * y + z
```

(continues on next page)

(continued from previous page)

```
        return w,

    def backward_cpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

As per the warning above, the `forward_cpu` method returns a tuple of single element. Note that all arrays appearing in CPU functions are `numpy.ndarray`. The forward function is straightforward; it unpacks the input tuple, computes the output, and packs it into a tuple. The backward function is a bit more complicated. Recall the rule of differentiation of multiplication. This example just implements the rule. Look at the return values, the function just packs the gradient of each input in the same order and returns them.

By just defining the core computation of forward and backward, *Function* class provides a chaining logic on it (i.e., storing the history of computation, etc.).

Note: Assuming we implement a (forward) function $y = f(x)$ which takes as input the vector $x \in \mathbb{R}^n$ and produces as output a vector $y \in \mathbb{R}^m$. Then the `backward` method has to compute

$$\lambda_i = \sum_{j=1}^{m} \frac{\partial y_j}{\partial x_i} \gamma_j \text{ for } i = 1 \dots n$$

where γ is the `grad_outputs`. Note, that the resulting vector λ must have the same shape as the arguments of the `forward` method.

Now let's define the corresponding GPU methods. You can easily predict that the methods we have to write are named *forward_gpu()* and *backward_gpu()*:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        ...

    def backward_cpu(self, inputs, grad_outputs):
        ...

    def forward_gpu(self, inputs):
        x, y, z = inputs
        w = x * y + z
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

In GPU methods, arrays are of type `cupy.ndarray`. We use arithmetic operators defined for this class. These operators implement the basic elementwise arithmetics.

You may find that the definitions of GPU methods are exactly same as those of CPU methods. In that case, we can reduce them to *forward()* and *backward()* methods.

```
class MulAdd(Function):
    def forward(self, inputs):
        x, y, z = inputs
        w = x * y + z
        return w,

    def backward(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

Since the `cupy.ndarray` class implements many methods of `numpy.ndarray`, we can write these unified methods in most cases.

The MulAdd function can be used as follows:

```
x = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
y = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
z = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
w = MulAdd()(x, y, z)
```

It looks a bit ugly: we have to explicitly instantiate MulAdd before applying it to variables. We also have to be careful that one instance of MulAdd must not be used multiple times, since it acts as a node in the computational graph. In Chainer, we often define a thin wrapper Python function that hide the instantiation:

```
def muladd(x, y, z):
    return MulAdd()(x, y, z)

w = muladd(x, y, z)
```

All functions under *chainer.functions* are implemented as wrapper functions like this.

Unified forward/backward methods with NumPy/CuPy functions

CuPy implements many functions that are compatible to those of NumPy. We can write unified forward/backward methods with them. Consider that we want to write a backprop-able function $f(x, y) = \exp(x) + \exp(y)$. We name it *ExpAdd* here. It can be written straight-forward as follows:

```
from chainer.backends import cuda

class ExpAdd(Function):
    def forward_cpu(self, inputs):
        x, y = inputs
        z = np.exp(x) + np.exp(y)
        return z,

    def backward_cpu(self, inputs, grad_outputs):
        x, y = inputs
        gz, = grad_outputs
```

(continues on next page)

(continued from previous page)

```
        gx = gz * np.exp(x)
        gy = gz * np.exp(y)
        return gx, gy

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y = inputs
        z = cupy.exp(x) + cupy.exp(y)
        return z,

    def backward_gpu(self, inputs, grad_outputs):
        cupy = cuda.cupy
        x, y = inputs
        gz, = grad_outputs

        gx = gz * cupy.exp(x)
        gy = gz * cupy.exp(y)
        return gx, gy

def expadd(x, y):
    return ExpAdd()(x, y)
```

Note: Here we used `chainer.backends.cuda.cupy` instead of directly accessing `cupy`. This is because the `cupy` module cannot be imported if the CUDA is not installed. In order to keep the implementation valid in non-CUDA environment, we have to defer the access to the `cupy` module. Note that the *`chainer.backends.cuda`* module can be imported even if the CUDA is not installed. Of course, the module in such environment is almost useless, but if the interpreter does not run through the code accessing CUDA-dedicated functions, the code is still valid.

The CPU and GPU implementations are almost same, except that `numpy` is replaced by `cupy` in GPU methods. We can unify these functions using the *`chainer.backend.get_array_module()`* function. This function accepts arbitrary number of arrays, and returns an appropriate module for them. See the following code:

```
class ExpAdd(Function):
    def forward(self, inputs):
        xp = backend.get_array_module(*inputs)
        x, y = inputs
        z = xp.exp(x) + xp.exp(y)
        return z,

    def backward(self, inputs, grad_outputs):
        xp = backend.get_array_module(*inputs)
        x, y = inputs
        gz, = grad_outputs

        gx = gz * xp.exp(x)
        gy = gz * xp.exp(y)
        return gx, gy

def expadd(x, y):
    return ExpAdd()(x, y)
```

Note that this code works correctly even if CUDA is not installed in the environment. If CUDA is not found, *`get_array_module()`* function always returns `numpy`. We often use the name `xp` for the variadic module name, which is analogous to the abbreviation `np` for NumPy and `cp` for CuPy.

Write an Elementwise Kernel Function

Let's turn back to the MulAdd example.

The GPU implementation of MulAdd as shown above is already fast and parallelized on GPU cores. However, it invokes two kernels during each of forward (`w = x * y + z`) and backward (`gx = y * gw` and `gy = x * gw`) computations. It might hurt performance, since the intermediate temporary arrays are read and written by possibly different GPU cores, which consumes much bandwidth. We can reduce the number of invocations by defining our own kernel. It also reduce the memory consumption.

Most functions only require elementwise operations like MulAdd. CuPy provides a useful tool to define elementwise kernels, the `cupy.ElementwiseKernel` class, and Chainer wraps it by `chainer.backends.cuda.elementwise()` function. Our MulAdd implementation can be improved as follows:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        ...

    def backward_cpu(self, inputs, grad_outputs):
        ...

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y, z = inputs
        w = cuda.elementwise(
            'float32 x, float32 y, float32 z',
            'float32 w',
            'w = x * y + z',
            'muladd_fwd')(x, y, z)
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx, gy = cuda.elementwise(
            'float32 x, float32 y, float32 gw',
            'float32 gx, float32 gy',
            '''
               gx = y * gw;
               gy = x * gw;
            ''',
            'muladd_bwd')(x, y, gw)

        gz = gw
        return gx, gy, gz
```

`chainer.backends.cuda.elementwise()` function accepts the essential implementation of the kernel function, and returns a kernel invocation function (actually, it returns `ElementwiseKernel` object, which is callable). In typical usage, we pass four arguments to this function as follows:

1. Input argument list. This is a comma-separated string each entry of which consists of a type specification and an argument name.
2. Output argument list in the same format as the input argument list.
3. Body of *parallel loop*. We can use the input/output argument names as an element of these arrays.
4. Name of the kernel function, which is shown in debuggers and profilers.

Above code is not compiled on every forward/backward computation thanks to two caching mechanisms provided by `chainer.backends.cuda.elementwise()`.

The first one is *binary caching*: `chainer.backends.cuda.elementwise()` function caches the compiled binary in the `$(HOME)/.cupy/kernel_cache` directory with a hash value of the CUDA code, and reuses it if the given code matches the hash value. This caching mechanism is actually implemented in CuPy.

The second one is *upload caching*: Given a compiled binary code, we have to upload it to the current GPU in order to execute it. `chainer.backends.cuda.elementwise()` function memoizes the arguments and the current device, and if it is called with the same arguments for the same device, it reuses the previously uploaded kernel code.

The above MulAdd code only works for float32 arrays. The `ElementwiseKernel` also supports the type-variadic kernel definition. In order to define variadic kernel functions, you can use *type placeholder* by placing a single character as type specifier:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        ...

    def backward_cpu(self, inputs, grad_outputs):
        ...

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y, z = inputs
        w = cuda.elementwise(
            'T x, T y, T z',
            'T w',
            'w = x * y + z',
            'muladd_fwd')(x, y, z)
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs

        gx, gy = cuda.elementwise(
            'T x, T y, T gw',
            'T gx, T gy',
            '''
               gx = y * gw;
               gy = x * gw;
            ''',
            'muladd_bwd')(x, y, gw)

        gz = gw
        return gx, gy, gz
```

The type placeholder `T` indicates an arbitrary data type that CuPy supports.

There are more functionalities on user-defined kernels in CuPy. See the CuPy documentation on user-defined kernels for more details.

2.5 Creating Models

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

Most neural network architectures contain multiple links. For example, a multi-layer perceptron consists of multiple linear layers. We can write complex procedures with parameters by combining multiple links like this:

```
>>> l1 = L.Linear(4, 3)
>>> l2 = L.Linear(3, 2)

>>> def my_forward(x):
...     h = l1(x)
...     return l2(h)
```

Here the `L` indicates the *links* module. A procedure with parameters defined in this way is hard to reuse. More Pythonic way is combining the links and procedures into a class:

```
>>> class MyProc(object):
...     def __init__(self):
...         self.l1 = L.Linear(4, 3)
...         self.l2 = L.Linear(3, 2)
...
...     def forward(self, x):
...         h = self.l1(x)
...         return self.l2(h)
```

In order to make it more reusable, we want to support parameter management, CPU/GPU migration, robust and flexible save/load features, etc. These features are all supported by the *Chain* class in Chainer. Then, what we have to do here is just define the above class as a subclass of Chain:

```
>>> class MyChain(Chain):
...     def __init__(self):
...         super(MyChain, self).__init__()
...         with self.init_scope():
...             self.l1 = L.Linear(4, 3)
...             self.l2 = L.Linear(3, 2)
...
...     def forward(self, x):
...         h = self.l1(x)
...         return self.l2(h)
```

It shows how a complex chain is constructed by simpler links. Links like `l1` and `l2` are called *child links* of `MyChain`. **Note that Chain itself inherits Link**. It means we can define more complex chains that hold `MyChain` objects as their child links.

Note: We often define a single forward method of a link by the `forward` operator. Such links and chains are callable and behave like regular functions of Variables.

Another way to define a chain is using the `ChainList` class, which behaves like a list of links:

```
>>> class MyChain2(ChainList):
...     def __init__(self):
...         super(MyChain2, self).__init__(
...             L.Linear(4, 3),
...             L.Linear(3, 2),
...         )
...
...     def forward(self, x):
...         h = self[0](x)
...         return self[1](h)
```

ChainList can conveniently use an arbitrary number of links, however if the number of links is fixed like in the above case, the Chain class is recommended as a base class.

2.6 Optimizer

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

From the previous guide on *Creating Models*, let's use the `MyChain` class:

```
>>> class MyChain(Chain):
...     def __init__(self):
...         super(MyChain, self).__init__()
...         with self.init_scope():
...             self.l1 = L.Linear(4, 3)
...             self.l2 = L.Linear(3, 2)
...
...     def forward(self, x):
...         h = self.l1(x)
...         return self.l2(h)
```

To tune parameters values to minimize loss, etc., we have to optimize them by the `Optimizer` class. It runs a numerical optimization algorithm on a given link. Many algorithms are implemented in the `optimizers` module. Here we use the simplest one, called Stochastic Gradient Descent (SGD):

```
>>> model = MyChain()
>>> optimizer = optimizers.SGD().setup(model)
```

The method `setup()` prepares for the optimization given a link.

Some parameter/gradient manipulations, e.g. weight decay and gradient clipping, can be done by setting *hook functions* to the optimizer. Hook functions are called after the gradient computation and right before the actual update of parameters. For example, we can set weight decay regularization by running the next line beforehand:

```
>>> optimizer.add_hook(chainer.optimizer_hooks.WeightDecay(0.0005))
```

Of course, you can write your own hook functions. It should be a function or a callable object.

There are two ways to use the optimizer. One is using it via `Trainer`, which we will see in the following sections. The other way is using it directly. We here review the latter case. To use the optimizer in an automated fashion, see the *Trainer* guide.

There are two further ways to use the optimizer directly. One is manually computing gradients and then calling the `update()` method with no arguments. Do not forget to clear the gradients beforehand!

```
>>> x = np.random.uniform(-1, 1, (2, 4)).astype(np.float32)
>>> model.cleargrads()
>>> # compute gradient here...
>>> loss = F.sum(model(chainer.Variable(x)))
>>> loss.backward()
>>> optimizer.update()
```

The other way is just passing a loss function to the `update()` method. In this case, `cleargrads()` is automatically called by the update method, so the user does not have to call it manually.

```
>>> def lossfun(arg1, arg2):
...     # calculate loss
...     loss = F.sum(model(arg1 - arg2))
...     return loss

>>> arg1 = np.random.uniform(-1, 1, (2, 4)).astype(np.float32)
>>> arg2 = np.random.uniform(-1, 1, (2, 4)).astype(np.float32)
>>> optimizer.update(lossfun, chainer.Variable(arg1), chainer.Variable(arg2))
```

See `chainer.Optimizer.update()` for the full specification.

2.7 Trainer

When we want to train neural networks, we have to run *training loops* that update the parameters many times. A typical training loop consists of the following procedures:

1. Iterations over training datasets
2. Preprocessing of extracted mini-batches
3. Forward/backward computations of the neural networks
4. Parameter updates
5. Evaluations of the current parameters on validation datasets
6. Logging and printing of the intermediate results

Chainer provides a simple yet powerful way to make it easy to write such training processes. The training loop abstraction mainly consists of two components:

- **Dataset abstraction**. It implements 1 and 2 in the above list. The core components are defined in the *dataset* module. There are also many implementations of datasets and iterators in *datasets* and *iterators* modules, respectively.
- **Trainer**. It implements 3, 4, 5, and 6 in the above list. The whole procedure is implemented by *Trainer*. The way to update parameters (3 and 4) is defined by *Updater*, which can be freely customized. 5 and 6 are implemented by instances of *Extension*, which appends an extra procedure to the training loop. Users can freely customize the training procedure by adding extensions. Users can also implement their own extensions.

2.8 Trainer Extensions

In this section, you will learn about the following topics:

- How to create your own trainer extension
  - by defining a simple function
  - *by defining a function decorated with @make_extension*
  - *by defining a class inherited from Extension class*

  In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

2.8.1 What is trainer Extension?

Extension is a callable object that takes a *Trainer* object as an argument. By adding an *Extension* to a *Trainer* using the *extend()* method, the *Extension* will be called according to the schedule specified by using a `trigger` object (See the details in *1. trigger*)

The *Trainer* object contains all information used in a training loop, e.g., models, optimizers, updaters, iterators, and datasets, etc. This makes it possible to change settings such as the learning rate of an optimizer.

2.8.2 Write a simple function

You can make a new *Extension* by writing a simple function which takes a *Trainer* object as its argument. For example, when you want to reduce the learning rate periodically during training, an `lr_drop` extension can be written as follows:

```
def lr_drop(trainer):
    trainer.updater.get_optimizer('main').lr *= 0.1
```

Then you can add this function to a *Trainer* object via *extend()* method.

```
trainer.extend(lr_drop, trigger=(10, 'epoch'))
```

It lowers the learning rate every 10 epochs by multiplying 0.1 with the current learning rate.

2.8.3 Write a function decorated with @make_extension

make_extension() is a decorator that adds some attributes to a given function. For example, the simple extension we created above can be written in this form:

```
@training.make_extension(trigger=(10, 'epoch'))
def lr_drop(trainer):
    trainer.updater.get_optimizer('main').lr *= 0.1
```

The difference between the above example and this is whether it has a default `trigger` or not. In the latter case, `lr_drop()` has its default `trigger` so that unless another `trigger` is specified via *extend()* method, the `trigger` specified in *make_extension()* is used by default. The code below acts the same as the former example, i.e., it reduces the learning rate every 10 epochs.

```
trainer.extend(lr_drop)
```

There are several attributes you can add using the *make_extension()* decorator.

1. trigger

`trigger` is an object that takes a *Trainer* object as an argument and returns a boolean value. If a tuple in the form `(period, unit)` is given as a trigger, it will be considered as an *IntervalTrigger* that invokes the extension every `period` `unit`. For example, when the given tuple is `(10, 'epoch')`, the extension will run every 10 epochs.

`trigger` can also be given to the *extend()* method that adds an extension to a *Trainer* object. The priority of `triggers` is as follows:

- When both *extend()* and a given *Extension* have `triggers`, the `trigger` given to *extend()* is used.
- When `None` is given to *extend()* as the `trigger` argument and a given *Extension* has `trigger`, the `trigger` given to the *Extension* is used.
- When both `trigger` attributes in *extend()* and *Extension* are `None`, the *Extension* will be fired every iteration.

See the details in the documentation of *get_trigger()* for more information.

2. default_name

An *Extension* is kept in a dictionary which is a property in a *Trainer*. This argument gives the name of the *Extension*. Users will see this name in the keys of the snapshot which is a dictionary generated by serialization.

3. priority

As a *Trainer* object can be assigned multiple *Extension* objects, the execution order is defined according to the following three values:

- `PRIORITY_WRITER`: The priority for extensions that write some records to the observation dictionary. It includes cases that the extension directly adds values to the observation dictionary, or the extension uses the chainer.report() function to report values to the observation dictionary. Extensions which write something to reporter should go first because other Extensions which read those values may be added.
- `PRIORITY_EDITOR`: The priority for extensions that edit the observation dictionary based on already reported values. Extensions which edit some values of reported ones should go after the extensions which write values to reporter but before extensions which read the final values.
- `PRIORITY_READER`: The priority for extensions that only read records from the observation dictionary. This is also suitable for extensions that do not use the observation dictionary at all. Extensions which read the reported values should be fired after all the extensions which have other priorities, e.g, `PRIORITY_WRITER` and `PRIORITY_EDITOR` because it should read the final values.

See the details in the documentation of *Trainer* for more information.

4. finalizer

You can specify a function to finalize the extension. It is called once at the end of the training loop, i.e., when *run()* has finished.

5. initializer

You can specify a function which takes a *Trainer* object as an argument to initialize the extension. It is called once before the training loop begins.

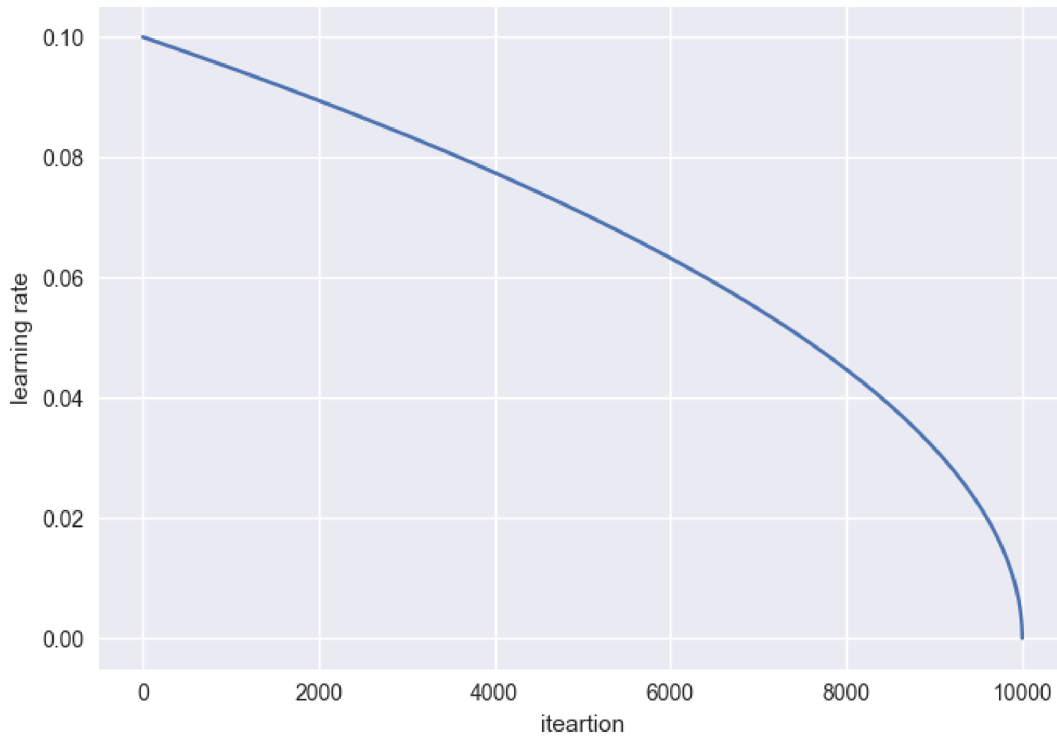
2.8.4 Write a class inherited from the Extension class

This is the way to define your own extension with the maximum degree of freedom. You can keep any values inside of the extension and serialize them.

As an example, let's make an extension that drops the learning rate polynomially. It calculates the learning rate by this equation:

$$\eta = \eta_{\rm init} \left(1 - \frac{t}{t_{\rm max}}\right)^{\rm power}$$

The learning rate will be dropped according to the curve below with ${\rm power} = 0.5$:




```
class PolynomialShift(training.Extension):

    def __init__(self, attr, power, stop_trigger, batchsize=None,
                 len_dataset=None):
        self._attr = attr
        self._power = power
        self._init = None
        self._t = 0
        self._last_value = 0

        if stop_trigger[1] == 'iteration':
            self._maxiter = stop_trigger[0]
        elif stop_trigger[1] == 'epoch':
            if batchsize is None or len_dataset is None:
                raise ValueError(
                    'When the unit of \'stop_trigger\' is \'epoch\', '
                    '\'batchsize\' and \'len_dataset\' should be '
                    'specified to calculate the maximum iteration.')
            n_iter_per_epoch = len_dataset / float(batchsize)
            self._maxiter = float(stop_trigger[0] * n_iter_per_epoch)

    def initialize(self, trainer):
        optimizer = trainer.updater.get_optimizer('main')
        # ensure that _init is set
        if self._init is None:
            self._init = getattr(optimizer, self._attr)

    def __call__(self, trainer):
```

(continues on next page)

(continued from previous page)

```
        self._t += 1

        optimizer = trainer.updater.get_optimizer('main')
        value = self._init * ((1 - (self._t / self._maxiter)) ** self._power)
        setattr(optimizer, self._attr, value)
        self._last_value = value

    def serialize(self, serializer):
        self._t = serializer('_t', self._t)
        self._last_value = serializer('_last_value', self._last_value)
        if isinstance(self._last_value, np.ndarray):
            self._last_value = self._last_value.item()
```

```
stop_trigger = (10000, 'iteration')
trainer.extend(PolynomialShift('lr', 0.5, stop_trigger))
```

This extension `PolynomialShift` takes five arguments.

- `attr`: The name of the optimizer property you want to update using this extension.
- `power`: The power of the above equation to calculate the learning rate.
- `stop_trigger`: The trigger given to the *Trainer* object to specify when to stop the training loop.
- `batchsize`: The training mini-batchsize.
- `len_dataset`: The length of the dataset, i.e., the number of data in the training dataset.

This extension calculates the number of iterations which will be performed during training by using `stop_trigger`, `batchsize`, and `len_dataset`, then stores it as a property `_maxiter`. This property will be used in the `__call__()` method to update the learning rate. The `initialize()` method obtains the initial learning rate from the optimizer given to the *Trainer* object. The `serialize()` method stores or recovers the properties, `_t` (number of iterations) and `_last_value` (the latest learning rate), belonging to this extension.

2.9 Using GPU(s) in Chainer

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

In this section, you will learn about the following topics:

- Relationship between Chainer and CuPy

- Basics of CuPy
- Single-GPU usage of Chainer
- Multi-GPU usage of model-parallel computing
- Multi-GPU usage of data-parallel computing

After reading this section, you will be able to:

- Use Chainer on a CUDA-enabled GPU
- Write model-parallel computing in Chainer
- Write data-parallel computing in Chainer

2.9.1 Relationship between Chainer and CuPy

Note: Even if you have CUDA installed in your environment, you have to install CuPy separately to use GPUs. See Working with Custom CUDA Installation for the way to set up CUDA support.

Chainer uses CuPy as its backend for GPU computation. In particular, the `cupy.ndarray` class is the GPU array implementation for Chainer. CuPy supports a subset of features of NumPy with a compatible interface. It enables us to write a common code for CPU and GPU. It also supports PyCUDA-like user-defined kernel generation, which enables us to write fast implementations dedicated to GPU.

Note: The *`chainer.backends.cuda`* module imports many important symbols from CuPy. For example, the cupy namespace is referred as `cuda.cupy` in the Chainer code. Note that the *`chainer.backends.cuda`* module can be imported even if CUDA is not installed.

Chainer uses a memory pool for GPU memory allocation. As shown in the previous sections, Chainer constructs and destructs many arrays during learning and evaluating iterations. It is not well suited for CUDA architecture, since memory allocation and release in CUDA (i.e. `cudaMalloc` and `cudaFree` functions) synchronize CPU and GPU computations, which hurts performance. In order to avoid memory allocation and deallocation during the computation, Chainer uses CuPy's memory pool as the standard memory allocator. Chainer changes the default allocator of CuPy to the memory pool, so user can use functions of CuPy directly without dealing with the memory allocator.

2.9.2 Basics of `cupy.ndarray`

See the documentation of CuPy for the basic usage of `cupy.ndarray`

CuPy is a GPU array backend that implements a subset of NumPy interface. The `cupy.ndarray` class is in its core, which is a compatible GPU alternative of `numpy.ndarray`. CuPy implements many functions on `cupy.ndarray` objects. See the reference for the supported subset of NumPy API. Understanding NumPy might help utilizing most features of CuPy. See the NumPy documentation for learning it.

The main difference of `cupy.ndarray` from `numpy.ndarray` is that the content is allocated on the device memory. The allocation takes place on the current device by default. The current device can be changed by `cupy.cuda.Device` object as follows:

```
with cupy.cuda.Device(1):
    x_on_gpu1 = cupy.array([1, 2, 3, 4, 5])
```

Most operations of CuPy is done on the current device. Be careful that it causes an error to process an array on a non-current device.

Chainer provides some convenient functions to automatically switch and choose the device. For example, the `chainer.backends.cuda.to_gpu()` function copies a `numpy.ndarray` object to a specified device:

```
x_cpu = np.ones((5, 4, 3), dtype=np.float32)
x_gpu = cuda.to_gpu(x_cpu, device=1)
```

It is equivalent to the following code using CuPy:

```
x_cpu = np.ones((5, 4, 3), dtype=np.float32)
with cupy.cuda.Device(1):
    x_gpu = cupy.array(x_cpu)
```

Moving a device array to the host can be done by `chainer.backends.cuda.to_cpu()` as follows:

```
x_cpu = cuda.to_cpu(x_gpu)
```

It is equivalent to the following code using CuPy:

```
with x_gpu.device:
    x_cpu = x_gpu.get()
```

Note: The *with* statements in these codes are required to select the appropriate CUDA device. If user uses only one device, these device switching is not needed. `chainer.backends.cuda.to_cpu()` and `chainer.backends.cuda.to_gpu()` functions automatically switch the current device correctly.

Chainer also provides a convenient function `chainer.backends.cuda.get_device_from_id()` and `chainer.backends.cuda.get_device_from_array()` to select a device. The former function accepts an integer or `None`. When `None` is given, it returns *a dummy device object*. Otherwise, it returns a corresponding device object. The latter function accepts CuPy array or NumPy array. When a NumPy array is given, it returns *a dummy device object*. Otherwise, it returns a corresponding device object to the give CuPy array. The dummy device object also supports *with* statements like the above example but does nothing. Here are some other examples:

```
cuda.get_device_from_id(1).use()
x_gpu1 = cupy.empty((4, 3), dtype=cupy.float32)

with cuda.get_device_from_id(1):
    x_gpu1 = cupy.empty((4, 3), dtype=cupy.float32)

with cuda.get_device_from_array(x_gpu1):
    y_gpu1 = x_gpu + 1
```

Since it accepts NumPy arrays, we can write a function that accepts both NumPy and CuPy arrays with correct device switching:

```
def add1(x):
    with cuda.get_device_from_array(x):
        return x + 1
```

The compatibility of CuPy with NumPy enables us to write CPU/GPU generic code. It can be made easy by the `chainer.backend.get_array_module()` function. This function returns the `numpy` or `cupy` module based on arguments. A CPU/GPU generic function is defined using it like follows:

```
# Stable implementation of log(1 + exp(x))
def softplus(x):
    xp = backend.get_array_module(x)
    return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

2.9.3 Run Neural Networks on a Single GPU

Single-GPU usage is very simple. What you have to do is transferring `Link` and input arrays to the GPU beforehand. In this subsection, the code is based on *our first MNIST example in this tutorial*.

A `Link` object can be transferred to the specified GPU using the `to_gpu()` method.

This time, we make the number of input, hidden, and output units configurable. The `to_gpu()` method also accepts a device ID like `model.to_gpu(0)`. In this case, the link object is transferred to the appropriate GPU device. The current device is used by default.

If we use `chainer.training.Trainer`, what we have to do is just let the updater know the device ID to send each mini-batch.

```
updater = training.updaters.StandardUpdater(train_iter, optimizer, device=0)
trainer = training.Trainer(updater, (20, 'epoch'), out='result')
```

We also have to specify the device ID for an evaluator extension as well.

```
trainer.extend(extensions.Evaluator(test_iter, model, device=0))
```

When we write down the training loop by hand, we have to transfer each mini-batch to the GPU manually:

```
model.to_gpu()
batchsize = 100
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
        x = Variable(cuda.to_gpu(x_train[indexes[i : i + batchsize]]))
        t = Variable(cuda.to_gpu(y_train[indexes[i : i + batchsize]]))
        optimizer.update(model, x, t)
```

2.9.4 Model-parallel Computation on Multiple GPUs

Parallelization of machine learning is roughly classified into two types called "model-parallel" and "data-parallel". Model-parallel means parallelizations of the computations inside the model. In contrast, data-parallel means parallelizations using data sharding. In this subsection, we show how to use the model-parallel approach on multiple GPUs in Chainer.

Recall the MNIST example. Now suppose that we want to modify this example by expanding the network to 6 layers with 2000 units each using two GPUs. In order to make multi-GPU computation efficient, we only make the two GPUs communicate at the third and sixth layer. The overall architecture looks like the following diagram:

```
(GPU0) input --+--> l1 --> l2 --> l3 --+--> l4 --> l5 --> l6 --+--> output
               |                       |                       |
(GPU1)         +--> l1 --> l2 --> l3 --+--> l4 --> l5 --> l6 --+
```

We can use the above MLP chain as following diagram:

```
(GPU0) input --+--> mlp1 --+--> mlp2 --+--> output
               |           |           |
(GPU1)         +--> mlp1 --+--> mlp2 --+
```

Let's write a link for the whole network.

```
class ParallelMLP(Chain):
    def __init__(self):
        super(ParallelMLP, self).__init__()
        with self.init_scope():
            # the input size, 784, is inferred
            self.mlp1_gpu0 = MLP(1000, 2000).to_gpu(0)
            self.mlp1_gpu1 = MLP(1000, 2000).to_gpu(1)

            # the input size, 2000, is inferred
            self.mlp2_gpu0 = MLP(1000, 10).to_gpu(0)
            self.mlp2_gpu1 = MLP(1000, 10).to_gpu(1)

    def forward(self, x):
        # assume x is on GPU 0
        z0 = self.mlp1_gpu0(x)
        z1 = self.mlp1_gpu1(F.copy(x, 1))

        # sync
        h0 = F.relu(z0 + F.copy(z1, 0))
        h1 = F.relu(z1 + F.copy(z0, 1))

        y0 = self.mlp2_gpu0(h0)
        y1 = self.mlp2_gpu1(h1)

        # sync
        y = y0 + F.copy(y1, 0)
        return y  # output is on GPU0
```

Recall that the `Link.to_gpu()` method returns the link itself. The `copy()` function copies an input variable to specified GPU device and returns a new variable on the device. The copy supports backprop, which just reversely transfers an output gradient to the input device.

Note: Above code is not parallelized on CPU, but is parallelized on GPU. This is because all the functions in the above code run asynchronously to the host CPU.

An almost identical example code can be found at examples/mnist/train_mnist_model_parallel.py.

2.9.5 Data-parallel Computation on Multiple GPUs with Trainer

Data-parallel computation is another strategy to parallelize online processing. In the context of neural networks, it means that a different device does computation on a different subset of the input data. In this subsection, we review the way to achieve data-parallel learning on two GPUs.

Suppose again our task is *the MNIST example*. This time we want to directly parallelize the three-layer network. The most simple form of data-parallelization is parallelizing the gradient computation for a distinct set of data. First, define a model and optimizer instances:

```
model = L.Classifier(MLP(1000, 10))  # the input size, 784, is inferred
optimizer = optimizers.SGD()
optimizer.setup(model)
```

Recall that the `MLP` link implements the multi-layer perceptron, and the *`Classifier`* link wraps it to provide a classifier interface. We used *`StandardUpdater`* in the previous example. In order to enable data-parallel computation with multiple GPUs, we only have to replace it with *`ParallelUpdater`*.

```
updater = training.updaters.ParallelUpdater(train_iter, optimizer,
                                            devices={'main': 0, 'second': 1})
```

The `devices` option specifies which devices to use in data-parallel learning. The device with name `'main'` is used as the main device. The original model is sent to this device, so the optimization runs on the main device. In the above example, the model is also cloned and sent to GPU 1. Half of each mini-batch is fed to this cloned model. After every backward computation, the gradient is accumulated into the main device, the parameter update runs on it, and then the updated parameters are sent to GPU 1 again.

See also the example code in examples/mnist/train_mnist_data_parallel.py.

2.9.6 Data-parallel Computation on Multiple GPUs without Trainer

We here introduce a way to write data-parallel computation without the help of *`Trainer`*. Most users can skip this section. If you are interested in how to write a data-parallel computation by yourself, this section should be informative. It is also helpful to, e.g., customize the *`ParallelUpdater`* class.

We again start from the MNIST example. At this time, we use a suffix like `_0` and `_1` to distinguish objects on each device. First, we define a model.

```
model_0 = L.Classifier(MLP(1000, 10))  # the input size, 784, is inferred
```

We want to make two copies of this instance on different GPUs. The *`Link.to_gpu()`* method runs in place, so we cannot use it to make a copy. In order to make a copy, we can use *`Link.copy()`* method.

```
model_1 = model_0.copy()
model_0.to_gpu(0)
model_1.to_gpu(1)
```

The *`Link.copy()`* method copies the link into another instance. *It just copies the link hierarchy*, and does not copy the arrays it holds.

Then, set up an optimizer:

```
optimizer = optimizers.SGD()
optimizer.setup(model_0)
```

Here we use the first copy of the model as *the master model*. Before its update, gradients of `model_1` must be aggregated to those of `model_0`.

Then, we can write a data-parallel learning loop as follows:

```
batchsize = 100
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
```

(continues on next page)

(continued from previous page)

```
        x_batch = x_train[indexes[i : i + batchsize]]
        y_batch = y_train[indexes[i : i + batchsize]]

        x0 = Variable(cuda.to_gpu(x_batch[:batchsize//2], 0))
        t0 = Variable(cuda.to_gpu(y_batch[:batchsize//2], 0))
        x1 = Variable(cuda.to_gpu(x_batch[batchsize//2:], 1))
        t1 = Variable(cuda.to_gpu(y_batch[batchsize//2:], 1))

        loss_0 = model_0(x0, t0)
        loss_1 = model_1(x1, t1)

        model_0.cleargrads()
        model_1.cleargrads()

        loss_0.backward()
        loss_1.backward()

        model_0.addgrads(model_1)
        optimizer.update()

        model_1.copyparams(model_0)
```

Do not forget to clear the gradients of both model copies! One half of the mini-batch is forwarded to GPU 0, the other half to GPU 1. Then the gradients are accumulated by the `Link.addgrads()` method. This method adds the gradients of a given link to those of the self. After the gradients are prepared, we can update the optimizer in usual way. Note that the update only modifies the parameters of `model_0`. So we must manually copy them to `model_1` using `Link.copyparams()` method.

Note: If the batch size used in one model remain the same, the scale of the gradient is roughly proportional to the number of models, when we aggregate gradients from all models by `chainer.Link.addgrads()`. So you need to adjust the batch size and/or learning rate of the optimizer accordingly.

Now you can use Chainer with GPUs. All examples in the examples directory support GPU computation, so please refer to them if you want to know more practices on using GPUs. In the next section, we will show how to define a differentiable (i.e. *backpropable*) function on Variable objects. We will also show there how to write a simple (elementwise) CUDA kernel using Chainer's CUDA utilities.

2.10 Type Checks

In this section, you will learn about the following things:

- Basic usage of type check
- Detail of type information
- Internal mechanism of type check
- More complicated cases
- Call functions
- Typical type check example

After reading this section, you will be able to:

- Write a code to check types of input arguments of your own functions

2.10.1 Basic usage of type check

When you call a function with an invalid type of array, you sometimes receive no error, but get an unexpected result by broadcasting. When you use CUDA with an illegal type of array, it causes memory corruption, and you get a serious error. These bugs are hard to fix. Chainer can check preconditions of each function, and helps to prevent such problems. These conditions may help a user to understand specification of functions.

Each implementation of `Function` has a method for type check, `check_type_forward()`. This function is called just before the `forward()` method of the `Function` class. You can override this method to check the condition on types and shapes of arguments.

`check_type_forward()` gets an argument `in_types`:

```
def check_type_forward(self, in_types):
  ...
```

`in_types` is an instance of `TypeInfoTuple`, which is a sub-class of `tuple`. To get type information about the first argument, use `in_types[0]`. If the function gets multiple arguments, we recommend to use new variables for readability:

```
x_type, y_type = in_types
```

In this case, `x_type` represents the type of the first argument, and `y_type` represents the second one.

We describe usage of `in_types` with an example. When you want to check if the number of dimension of `x_type` equals to `2`, write this code:

```
utils.type_check.expect(x_type.ndim == 2)
```

When this condition is true, nothing happens. Otherwise this code throws an exception, and the user gets a message like this:

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType: Expect: in_types[0].ndim == 2
Actual: 3 != 2
```

This error message means that "`ndim` of the first argument expected to be `2`, but actually it is `3`".

2.10.2 Detail of type information

You can access three information of `x_type`.

- `.shape` is a tuple of ints. Each value is size of each dimension.
- `.ndim` is `int` value representing the number of dimensions. Note that `ndim == len(shape)`
- `.dtype` is `numpy.dtype` representing data type of the value.

You can check all members. For example, the size of the first dimension must be positive, you can write like this:

```
utils.type_check.expect(x_type.shape[0] > 0)
```

You can also check data types with `.dtype`:

```
utils.type_check.expect(x_type.dtype == np.float64)
```

And an error is like this:

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType: Expect: in_types[0].dtype == <class 'numpy.
↪float64'>
Actual: float32 != <class 'numpy.float64'>
```

You can also check `kind` of `dtype`. This code checks if the type is floating point

```
utils.type_check.expect(x_type.dtype.kind == 'f')
```

You can compare between variables. For example, the following code checks if the first argument and the second argument have the same length:

```
utils.type_check.expect(x_type.shape[1] == y_type.shape[1])
```

2.10.3 Internal mechanism of type check

How does it show an error message like `"in_types[0].ndim == 2"`? If `x_type` is an object containing `ndim` member variable, we cannot show such an error message because this equation is evaluated as a boolean value by Python interpreter.

Actually `x_type` is a *`Expr`* objects, and doesn't have a `ndim` member variable itself. *`Expr`* represents a syntax tree. `x_type.ndim` makes a *`Expr`* object representing `(getattr, x_type, 'ndim')`. `x_type.ndim == 2` makes an object like `(eq, (getattr, x_type, 'ndim'), 2)`. *`expect()`* gets a *`Expr`* object and evaluates it. When it is `True`, it causes no error and shows nothing. Otherwise, this method shows a readable error message.

If you want to evaluate a *`Expr`* object, call *`eval()`* method:

```
actual_type = x_type.eval()
```

`actual_type` is an instance of *`TypeInfo`*, while `x_type` is an instance of *`Expr`*. In the same way, `x_type.shape[0].eval()` returns an int value.

2.10.4 More powerful methods

`Expr` class is more powerful. It supports all mathematical operators such as + and *. You can write a condition that the first dimension of `x_type` is the first dimension of `y_type` times four:

```
utils.type_check.expect(x_type.shape[0] == y_type.shape[0] * 4)
```

When `x_type.shape[0] == 3` and `y_type.shape[0] == 1`, users can get the error message below:

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType: Expect: in_types[0].shape[0] == in_types[1].
↪shape[0] * 4
Actual: 3 != 4
```

To compare a member variable of your function, wrap a value with *`Variable`* to show readable error message:

```
x_type.shape[0] == utils.type_check.Variable(self.in_size, "in_size")
```

This code can check the equivalent condition below:

```
x_type.shape[0] == self.in_size
```

However, the latter condition doesn't know the meaning of this value. When this condition is not satisfied, the latter code shows unreadable error message:

```
chainer.utils.type_check.InvalidType: Expect: in_types[0].shape[0] == 4  # what does
↪'4' mean?
Actual: 3 != 4
```

Note that the second argument of *utils.type_check.Variable* is only for readability.

The former shows this message:

```
chainer.utils.type_check.InvalidType: Expect: in_types[0].shape[0] == in_size  # OK,␣
↪`in_size` is a value that is given to the constructor
Actual: 3 != 4  # You can also check actual value here
```

2.10.5 Call functions

How to check summation of all values of shape? *Expr* also supports function call:

```
sum = utils.type_check.Variable(np.sum, 'sum')
utils.type_check.expect(sum(x_type.shape) == 10)
```

Why do we need to wrap the function `numpy.sum` with *utils.type_check.Variable*? `x_type.shape` is not a tuple but an object of *Expr* as we have seen before. Therefore, `numpy.sum(x_type.shape)` fails. We need to evaluate this function lazily.

The above example produces an error message like this:

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType: Expect: sum(in_types[0].shape) == 10
Actual: 7 != 10
```

2.10.6 More complicated cases

How to write a more complicated condition that can't be written with these operators? You can evaluate *Expr* and get its result value with *eval()* method. Then check the condition and show warning message by hand:

```
x_shape = x_type.shape.eval()  # get actual shape (int tuple)
if not more_complicated_condition(x_shape):
    expect_msg = 'Shape is expected to be ...'
    actual_msg = 'Shape is ...'
    raise utils.type_check.InvalidType(expect_msg, actual_msg)
```

Please write a readable error message. This code generates the following error message:

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType: Expect: Shape is expected to be ...
Actual: Shape is ...
```

2.10.7 Typical type check example

We show a typical type check for a function.

First check the number of arguments:

```
utils.type_check.expect(in_types.size() == 2)
```

`in_types.size()` returns a *Expr* object representing the number of arguments. You can check it in the same way.

And then, get each type:

```
x_type, y_type = in_types
```

Don't get each value before checking `in_types.size()`. When the number of argument is illegal, `type_check.expect` might output unuseful error messages. For example, this code doesn't work when the size of `in_types` is 0:

```
utils.type_check.expect(
  in_types.size() == 2,
  in_types[0].ndim == 3,
)
```

After that, check each type:

```
utils.type_check.expect(
  x_type.dtype == np.float32,
  x_type.ndim == 3,
  x_type.shape[1] == 2,
)
```

The above example works correctly even when `x_type.ndim == 0` as all conditions are evaluated lazily.

2.11 Serializers – saving and loading

Serializer is a simple interface to serialize or deserialize an object. *Link*, *Optimizer*, and *Trainer* support serialization.

Concrete serializers are defined in the *serializers* module. It supports NumPy NPZ and HDF5 formats.

For example, we can serialize a link object into NPZ file by the *save_npz()* function:

Assuming we have defined a `model`:

```
>>> from chainer import serializers
>>> serializers.save_npz('my.model', model)
```

This saves the parameters of `model` into the file `'my.model'` in NPZ format. The saved model can be read back from `my.model` back into `model` by the *load_npz()* function:

```
>>> serializers.load_npz('my.model', model)
```

Note: Note that only the parameters and the *persistent values* are serialized by this serialization code. Other attributes are not saved automatically. You can register arrays, scalars, or any serializable objects as persistent values by the `add_persistent()` method. The registered values can be accessed by attributes of the name passed to the add_persistent method.

The state of an optimizer can also be saved by the same functions:

```
>>> serializers.save_npz('my.state', optimizer)
>>> serializers.load_npz('my.state', optimizer)
```

Note: Note that serialization of optimizer only saves its internal states including number of iterations, momentum vectors of MomentumSGD, etc. It does not save the parameters and persistent values of the target link. We have to explicitly save the target link with the optimizer to resume the optimization from saved states. This can be done by saving the entire `Trainer` object, like this:

```
>>> serializers.save_npz('my.state', trainer)
```

Support of the HDF5 format is enabled if the h5py package is installed. Serialization and deserialization with the HDF5 format are almost identical to those with the NPZ format; just replace `save_npz()` and `load_npz()` by `save_hdf5()` and `load_hdf5()`, respectively.

2.12 Customize your own logging

In this section, you will learn about the following things:

- What is `chainer.Reporter`?
- How to report logging with `chainer.Reporter`?
- The naming rule for the reported values.

After reading this section, you will be able to:

- Write your own report.

2.12.1 What is Reporter?

`chainer.Reporter` is used to collect values that users want to watch. The reporter object manipulates a dictionary from value names to the actually observed values. We call this dictionary as *observation*.

See the following example:

```
>>> from chainer import Reporter, report, report_scope
>>>
>>> reporter = Reporter()
>>> observer = object()  # it can be an arbitrary (reference) object
>>> reporter.add_observer('my_observer:', observer)
>>> observation = {}
>>> with reporter.scope(observation):
```

(continues on next page)

(continued from previous page)

```
...     reporter.report({'x': 1}, observer)
...
>>> observation
{'my_observer:/x': 1}
```

When a value is passed to the `reporter`, an object called `observer` can be optionally attached. In this case, the name of the `observer` is added as the prefix of the value name. The `observer` name should be registered beforehand. Using `reporter.scope`, you can select which `observation` to save the observed values.

There are also a global API `chainer.report()`, which reports observed values with the current reporter object. In this case, *current* means which `with` statement scope the current code line is in. This function calls the `Reporter.report()` method of the current reporter.

```
>>> observation = {}
>>> with reporter.scope(observation):
...     report({'x': 1}, observer)
...
>>> observation
{'my_observer:/x': 1}
```

2.12.2 Use report in Chain or Link

The most important application of `Reporter` is to report observed values from each `Link` or `Chain` in the training and validation procedures.

But, how to report the observed values from each link or chain? Shold we prepare the `Reporter`? No, you only need to call `report()` in chain or link, because `Trainer` and some extensions prepare their own `Reporter` object with the hierarchy of the target link registered as observers. We can use `report()` function inside any links and chains to report the observed values (e.g., training loss, accuracy, activation statistics, etc.).

See the following example:

```
>>> class Classifier(Chain):
...     def __init__(self, predictor):
...         super(Classifier, self).__init__()
...         with self.init_scope():
...             self.predictor = predictor
...
...     def forward(self, x, t):
...         y = self.predictor(x)
...         loss = F.softmax_cross_entropy(y, t)
...         accuracy = F.accuracy(y, t)
...         report({'loss': loss, 'accuracy': accuracy}, self)
...         return loss
...
```

If the link is named `'main'` in the hierarchy (which is the default name of the target link in the `StandardUpdater`), these reported values are named `'main/loss'` and `'main/accuracy'`. If these values are reported inside the `Evaluator` extension, `'validation/'` is added at the head of the link name, thus the item names are changed to `'validation/main/loss'` and `'validation/main/accuracy'` (`'validation'` is the default name of the Evaluator extension).

2.12.3 Naming rule for the reported values

So, you know almost everything about *Reporter*. However, there is one more thing. It is what is the naming rule for the reported values, especially when the values are reported from a link that is not the root of the link hierarchy.

As we explained in the previous section, the root of links is named as `'main'` by the the `StandardUpdater` and the names of reported values in the root have the prefix `'main/'`. When the values are reported from a link that is not the root of the link hierarchy, the prefix of the names are determined by the link hierarchy, or *namedlinks()*.

See the following example:

```
>>> class MLP(Chain):
...     def __init__(self, n_units, n_out):
...         super(MLP, self).__init__()
...         with self.init_scope():
...             # the size of the inputs to each layer will be inferred
...             self.l1 = L.Linear(None, n_units)  # n_in -> n_units
...             self.l2 = L.Linear(None, n_units)  # n_units -> n_units
...             self.l3 = L.Linear(None, n_out)    # n_units -> n_out
...
...     def forward(self, x):
...         h1 = F.relu(self.l1(x))
...         h2 = F.relu(self.l2(h1))
...         y = self.l3(h2)
...         report({'sum_y': F.sum(y)}, self)
...         return y
...
>>> model = Classifier(MLP(100, 10))
>>> for name, observer in model.namedlinks(skipself=True):
...     print(name)
/predictor
/predictor/l1
/predictor/l2
/predictor/l3
```

You can get the parameters of the link hierarchy by *namedlinks()*. In this example, we report `'loss'` and `'accuracy'` in the root of links, and `'sum_y'` in the link of `'/predictor'`. So, you can access the reported values by `'main/accuracy'`, `'main/accuracy'`, and `'main/predictor/sum_y'`.

See what we explained is correct:

```
>>> train, test = datasets.get_mnist()
>>> train_iter = iterators.SerialIterator(train, batch_size=100, shuffle=True)
>>> test_iter = iterators.SerialIterator(test, batch_size=100, repeat=False,␣
↪shuffle=False)
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model)
>>> updater = training.StandardUpdater(train_iter, optimizer)
>>> trainer = training.Trainer(updater, (1, 'epoch'), out='result')
>>> trainer.extend(extensions.Evaluator(test_iter, model))
>>> trainer.extend(extensions.LogReport())
>>> trainer.extend(extensions.PrintReport(
...     ['epoch', 'main/accuracy', 'main/loss', 'main/predictor/sum_y', 'validation/
↪main/accuracy']))
>>> trainer.run()
epoch       main/accuracy  main/loss   main/predictor/sum_y  validation/main/accuracy
1           0.662317       1.38345     47.9927               0.8498
```

CHAPTER

THREE

NEURAL NET EXAMPLES

3.1 MNIST using Trainer

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

By using *Trainer*, you don't need to write the training loop explicitly any more. Furthermore, Chainer provides many useful extensions that can be used with *Trainer* to visualize your results, evaluate your model, store and manage log files more easily.

This example will show how to use the *Trainer* to train a fully-connected feed-forward neural network on the MNIST dataset.

Note: If you would like to know how to write a training loop without using the *Trainer*, please check *MNIST with a Manual Training Loop* instead of this tutorial.

3.1.1 1. Prepare the dataset

Load the MNIST dataset, which contains a training set of images and class labels as well as a corresponding test set.

```
from chainer.datasets import mnist

train, test = mnist.get_mnist()
```

Note: You can use a Python list as a dataset. That's because *Iterator* can take any object as a dataset whose elements can be accessed via `[]` accessor and whose length can be obtained with `len()` function. For example,

```
train = [(x1, t1), (x2, t2), ...]
```

a list of tuples like this can be used as a dataset.

There are many utility dataset classes defined in *datasets*. It is recommended that you utilize them in the actual applications.

For example, if your dataset consists of a number of image files, it would take a large amount of memory to load those data into a list like above. In that case, you can use *ImageDataset*, which just keeps the paths to image files. The actual image data will be loaded from the disk when the corresponding element is requested via `[]` accessor. Until then, no images are loaded to the memory to reduce memory use.

3.1.2 2. Prepare the dataset iterations

Iterator creates a mini-batch from the given dataset.

```
batchsize = 128

train_iter = iterators.SerialIterator(train, batchsize)
test_iter = iterators.SerialIterator(test, batchsize, False, False)
```

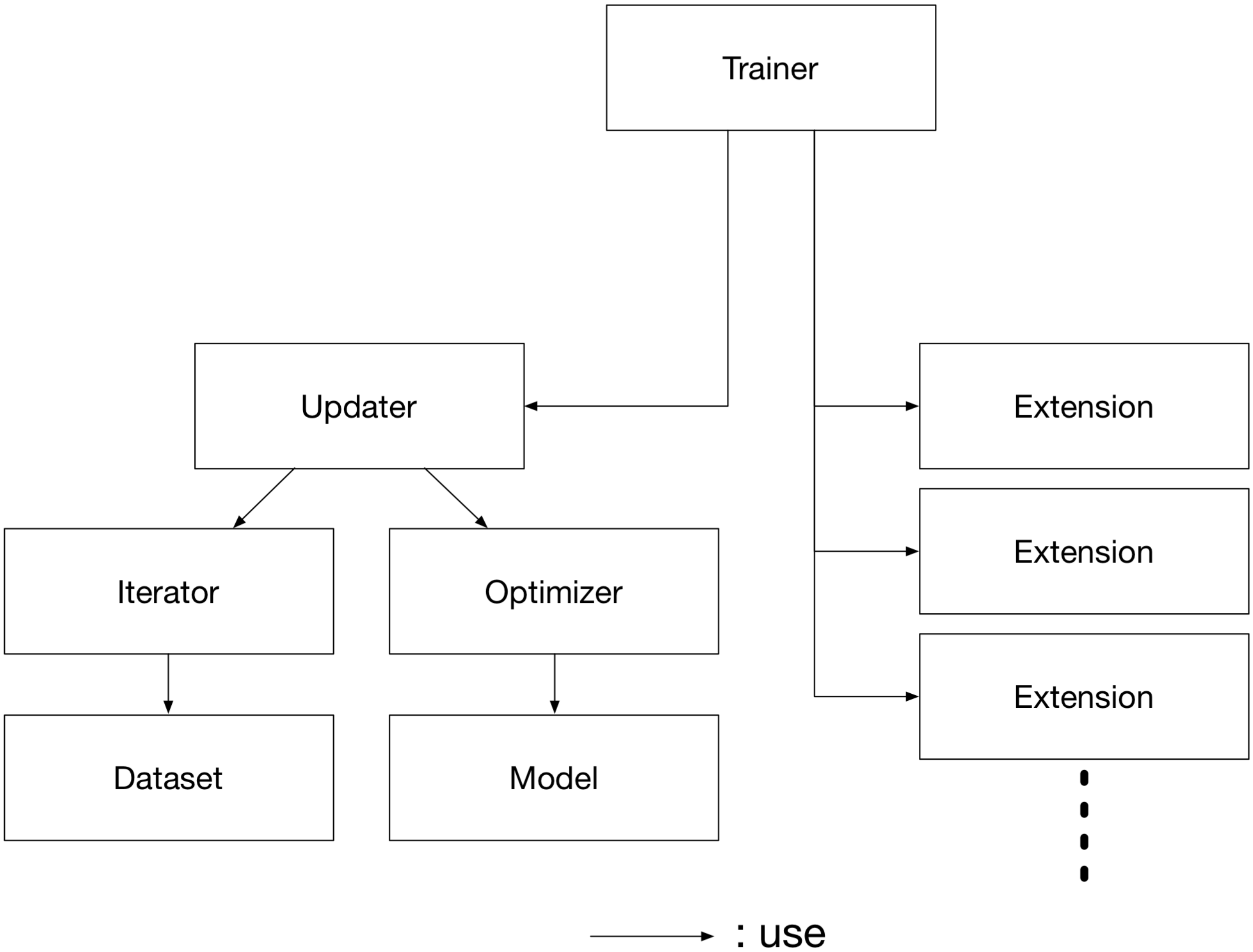
3.1.3 3. Prepare the model

Here, we are going to use the same model as the one defined in *MNIST with a Manual Training Loop*.

```
class MLP(Chain):

    def __init__(self, n_mid_units=100, n_out=10):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_mid_units)
            self.l2 = L.Linear(None, n_mid_units)
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

gpu_id = 0  # Set to -1 if you use CPU

model = MLP()
if gpu_id >= 0:
    model.to_gpu(gpu_id)
```

3.1.4 4. Prepare the Updater

Trainer is a class that holds all of the necessary components needed for training. The main components are shown below.



Basically, all you need to pass to `Trainer` is an `Updater`. However, `Updater` contains an `Iterator` and `Optimizer`. Since `Iterator` can access the dataset and `Optimizer` has references to the model, `Updater` can access to the model to update its parameters.

So, `Updater` can perform the training procedure as shown below:

1. Retrieve the data from dataset and construct a mini-batch (`Iterator`)
2. Pass the mini-batch to the model and calculate the loss
3. Update the parameters of the model (`Optimizer`)

Now let's create the `Updater` object !

```
max_epoch = 10

# Wrap your model by Classifier and include the process of loss calculation within␣
↪your model.
# Since we do not specify a loss function here, the default 'softmax_cross_entropy'␣
↪is used.
model = L.Classifier(model)

# selection of your optimizing method
optimizer = optimizers.MomentumSGD()

# Give the optimizer a reference to the model
```

(continues on next page)

(continued from previous page)

```
optimizer.setup(model)

# Get an updater that uses the Iterator and Optimizer
updater = training.updaters.StandardUpdater(train_iter, optimizer, device=gpu_id)
```

Note: Here, the model defined above is passed to `Classifier` and changed to a new `Chain`. `Classifier`, which in fact inherits from the `Chain` class, keeps the given `Chain` model in its `predictor` attribute. Once you give the input data and the corresponding class labels to the model by the `()` operator,

1. `forward()` of the model is invoked. The data is then given to `predictor` to obtain the output `y`.
2. Next, together with the given labels, the output `y` is passed to the loss function which is determined by `lossfun` argument in the constructor of `Classifier`.
3. The loss is returned as a `Variable`.

In `Classifier`, the `lossfun` is set to `softmax_cross_entropy()` as default.

`StandardUpdater` is the simplest class among several updaters. There are also the `ParallelUpdater` and the `MultiprocessParallelUpdater` to utilize multiple GPUs. The `MultiprocessParallelUpdater` uses the NVIDIA NCCL library, so you need to install NCCL and re-install CuPy before using it.

3.1.5 5. Setup Trainer

Lastly, we will setup `Trainer`. The only requirement for creating a `Trainer` is to pass the `Updater` object that we previously created above. You can also pass a `stop_trigger` to the second trainer argument as a tuple like `(length, unit)` to tell the trainer when to stop the training. The `length` is given as an integer and the `unit` is given as a string which should be either `epoch` or `iteration`. Without setting `stop_trigger`, the training will never be stopped.

```
# Setup a Trainer
trainer = training.Trainer(updater, (max_epoch, 'epoch'), out='mnist_result')
```

The `out` argument specifies an output directory used to save the log files, the image files of plots to show the time progress of loss, accuracy, etc. when you use `PlotReport` extension. Next, we will explain how to display or save those information by using trainer `Extension`.

3.1.6 6. Add Extensions to the Trainer object

The `Trainer` extensions provide the following capabilities:

- Save log files automatically (`LogReport`)
- Display the training information to the terminal periodically (`PrintReport`)
- Visualize the loss progress by plotting a graph periodically and save it as an image file (`PlotReport`)
- Automatically serialize the state periodically (`snapshot()` / `snapshot_object()`)
- Display a progress bar to the terminal to show the progress of training (`ProgressBar`)
- Save the model architecture as a Graphviz's dot file (`DumpGraph()`)

To use these wide variety of tools for your training task, pass `Extension` objects to the `extend()` method of your `Trainer` object.

```python
from chainer.training import extensions

trainer.extend(extensions.LogReport())
trainer.extend(extensions.snapshot(filename='snapshot_epoch-{.updater.epoch}'))
trainer.extend(extensions.snapshot_object(model.predictor, filename='model_epoch-{.
↪updater.epoch}'))
trainer.extend(extensions.Evaluator(test_iter, model, device=gpu_id))
trainer.extend(extensions.PrintReport(['epoch', 'main/loss', 'main/accuracy',
↪'validation/main/loss', 'validation/main/accuracy', 'elapsed_time']))
trainer.extend(extensions.PlotReport(['main/loss', 'validation/main/loss'], x_key=
↪'epoch', file_name='loss.png'))
trainer.extend(extensions.PlotReport(['main/accuracy', 'validation/main/accuracy'], x_
↪key='epoch', file_name='accuracy.png'))
trainer.extend(extensions.DumpGraph('main/loss'))
```

`LogReport`

Collect `loss` and `accuracy` automatically every `epoch` or `iteration` and store the information under the `log` file in the directory specified by the `out` argument when you create a *`Trainer`* object.

`snapshot()`

The *`snapshot()`* method saves the *`Trainer`* object at the designated timing (default: every epoch) in the directory specified by `out`. The *`Trainer`* object, as mentioned before, has an *`Updater`* which contains an *`Optimizer`* and a model inside. Therefore, as long as you have the snapshot file, you can use it to come back to the training or make inferences using the previously trained model later.

`snapshot_object()`

However, when you keep the whole *`Trainer`* object, in some cases, it is very tedious to retrieve only the inside of the model. By using *`snapshot_object()`*, you can save the particular object (in this case, the model wrapped by *`Classifier`*) as a separate snapshot. *`Classifier`* is a *`Chain`* object which keeps the model that is also a *`Chain`* object as its `predictor` property, and all the parameters are under the `predictor`, so taking the snapshot of `predictor` is enough to keep all the trained parameters.

This is a list of commonly used trainer extensions:

`LogReport` This extension collects the loss and accuracy values every epoch or iteration and stores in a log file. The log file will be located under the output directory (specified by `out` argument of the *`Trainer`* object).

`snapshot()` This extension saves the *`Trainer`* object at the designated timing (defaut: every epoch) in the output directory. The *`Trainer`* object, as mentioned before, has an *`Updater`* which contains an *`Optimizer`* and a model inside. Therefore, as long as you have the snapshot file, you can use it to come back to the training or make inferences using the previously trained model later.

`snapshot_object()` *`snapshot()`* extension above saves the whole *`Trainer`* object. However, in some cases, it is tedious to retrieve only the inside of the model. By using *`snapshot_object()`*, you can save the particular object (in the example above, the model wrapped by *`Classifier`*) as a separeted snapshot. Taking the snapshot of `predictor` is enough to keep all the trained parameters, because *`Classifier`* (which is a subclass of *`Chain`*) keeps the model as its `predictor` property, and all the parameters are under this property.

`DumpGraph()` This extension saves the structure of the computational graph of the model. The graph is saved in Graphviz dot format under the output directory of the *`Trainer`*.

Evaluator *Iterator*s that use the evaluation dataset and the model object are required to use *Evaluator* extension. It evaluates the model using the given dataset (typically it's a validation dataset) at the specified timing interval.

PrintReport This extension outputs the spcified values to the standard output.

PlotReport This extension plots the values specified by its arguments and saves it as a image file.

This is not an exhaustive list of built-in extensions. Please take a look at *Extensions* for more of them.

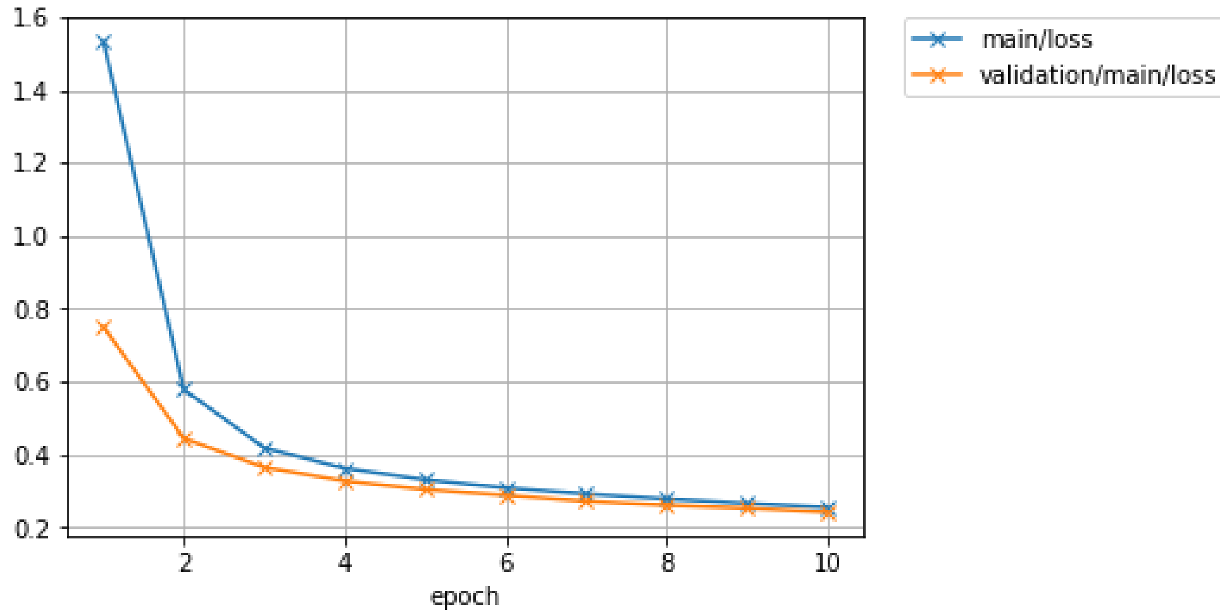
3.1.7 7. Start Training

Just call `run()` method from `Trainer` object to start training.

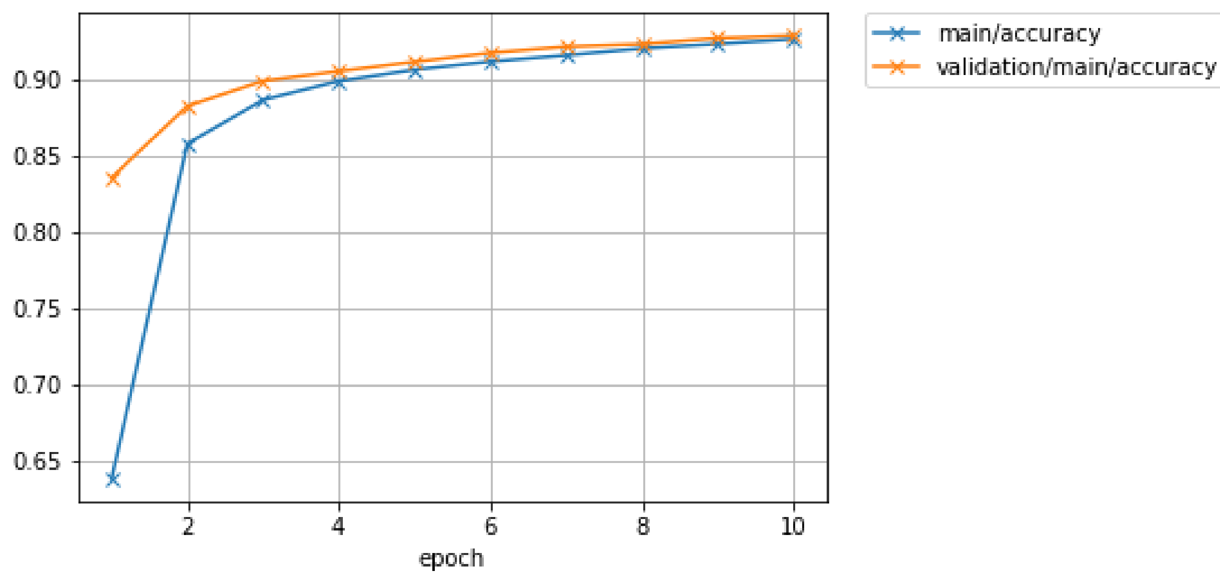
```
trainer.run()
```

```
epoch       main/loss   main/accuracy  validation/main/loss  validation/main/accuracy␣
↪ elapsed_time
1           1.53241     0.638409       0.74935               0.835839                ␣
↪ 4.93409
2           0.578334    0.858059       0.444722              0.882812                ␣
↪ 7.72883
3           0.418569    0.886844       0.364943              0.899229                ␣
↪ 10.4229
4           0.362342    0.899089       0.327569              0.905558                ␣
↪ 13.148
5           0.331067    0.906517       0.304399              0.911788                ␣
↪ 15.846
6           0.309019    0.911964       0.288295              0.917722                ␣
↪ 18.5395
7           0.292312    0.916128       0.272073              0.921776                ␣
↪ 21.2173
8           0.278291    0.92059        0.261351              0.923457                ␣
↪ 23.9211
9           0.266266    0.923541       0.253195              0.927314                ␣
↪ 26.6612
10          0.255489    0.926739       0.242415              0.929094                ␣
↪ 29.466
```

Let's see the plot of loss progress saved in the `mnist_result` directory.

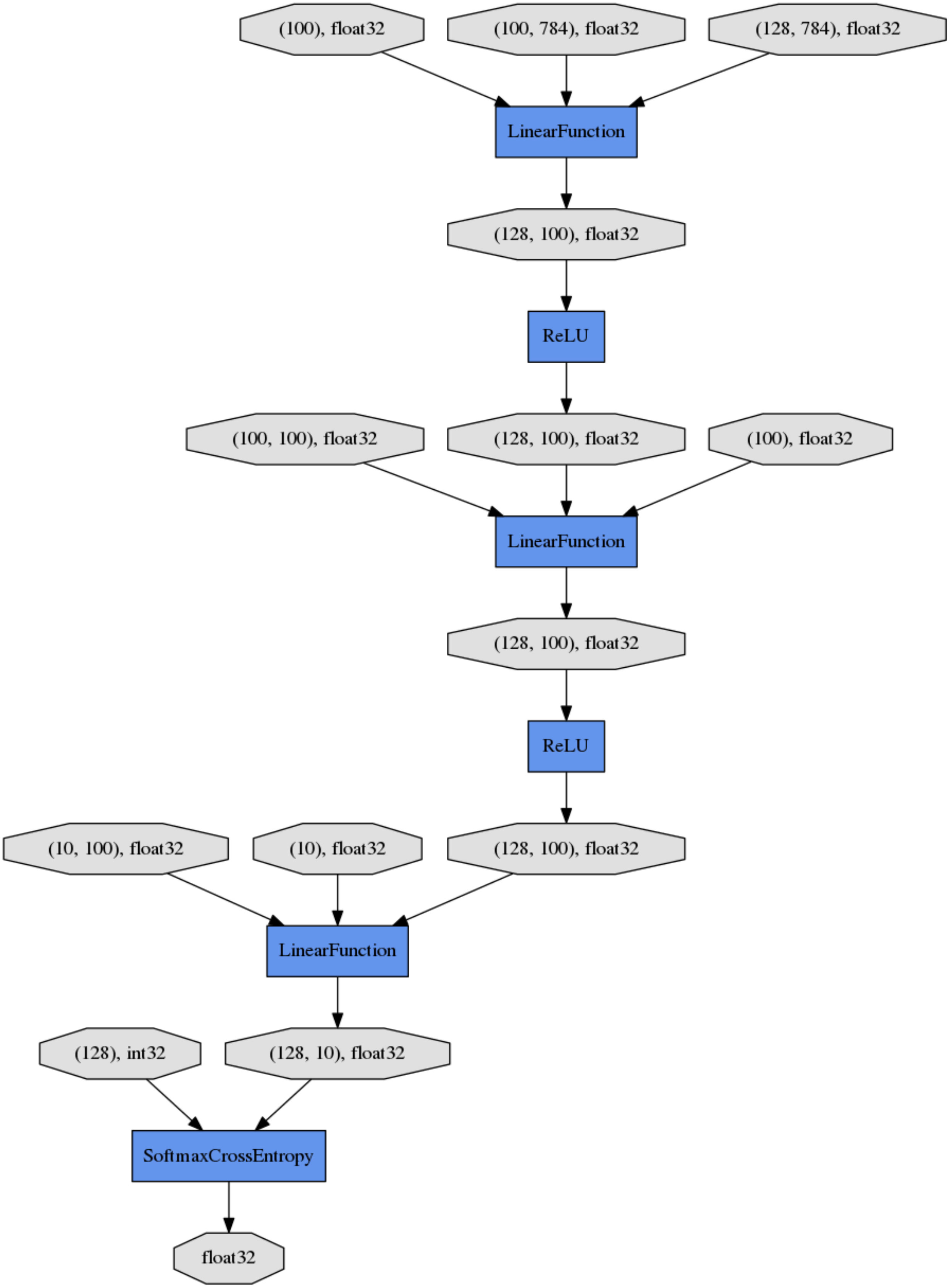


How about the accuracy?



Furthermore, let's visualize the computational graph saved with `DumpGraph()` using Graphviz.

```
% dot -Tpng mnist_result/cg.dot -o mnist_result/cg.png
```

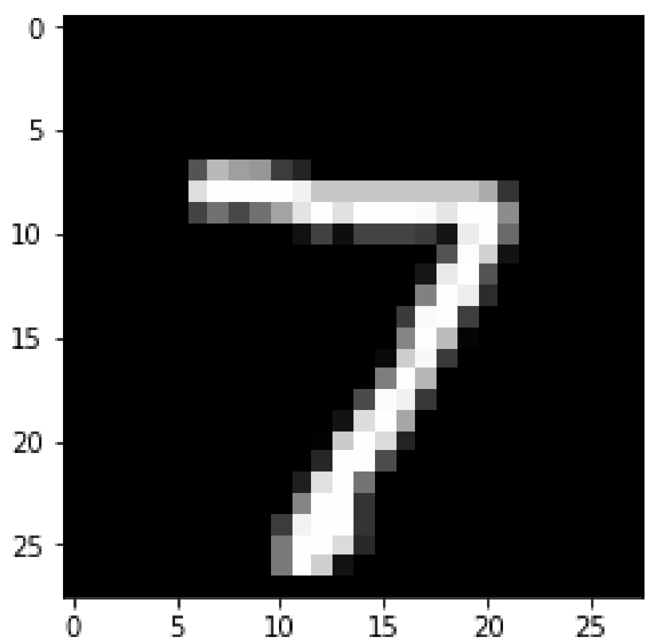

(100), float32
(100, 784), float32
(128, 784), float32
LinearFunction
(128, 100), float32
ReLU
(100, 100), float32
(128, 100), float32
(100), float32
LinearFunction
(128, 100), float32
ReLU
(10, 100), float32
(10), float32
(128, 100), float32
LinearFunction
(128), int32
(128, 10), float32
SoftmaxCrossEntropy
float32

From the top to the bottom, you can see the data flow in the computational graph. It basically shows how data and parameters are passed to the *Function*s.

3.1.8 8. Evaluate a pre-trained model

Evaluation using the snapshot of a model is as easy as what explained in the *MNIST with a Manual Training Loop*.

```
import matplotlib.pyplot as plt

model = MLP()
serializers.load_npz('mnist_result/model_epoch-10', model)

# Show the output
x, t = test[0]
plt.imshow(x.reshape(28, 28), cmap='gray')
plt.show()
print('label:', t)

y = model(x[None, ...])

print('predicted_label:', y.array.argmax(axis=1)[0])
```




```
label: 7
predicted_label: 7
```

The prediction looks correct. Success!

3.2 MNIST with a Manual Training Loop

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

In this tutorial section, we will learn how to train a deep neural network to classify images of hand-written digits in the popular MNIST dataset. This dataset contains 50,000 training examples and 10,000 test examples. Each example is a set of a 28 x 28 greyscale image and a corresponding class label. Since the digits from 0 to 9 are used, there are 10 classes for the labels.

Chainer provides a feature called `Trainer` that can simplify the training procedure of your model. However, it is also good to know how the training works in Chainer before starting to use the useful `Trainer` class that hides the actual processes. Writing your own training loop can be useful for learning how `Trainer` works or for implementing features not included in the standard trainer.

The complete training procedure consists of the following steps:

1. *Prepare a dataset*
2. *Create a dataset iterator*
3. *Define a network*
4. *Select an optimization algorithm*
5. *Write a training loop*
    a. Retrieve a set of examples (mini-batch) from the training dataset.
    b. Feed the mini-batch to your network.
    c. Run a forward pass of the network and compute the loss.
    d. Just call the `backward()` method from the loss `Variable` to compute the gradients for all trainable parameters.
    e. Run the optimizer to update those parameters.
6. *Save the trained model*
7. *Perform classification by the saved model* and check the network performance on validation/test sets.

3.2.1 1. Prepare a dataset

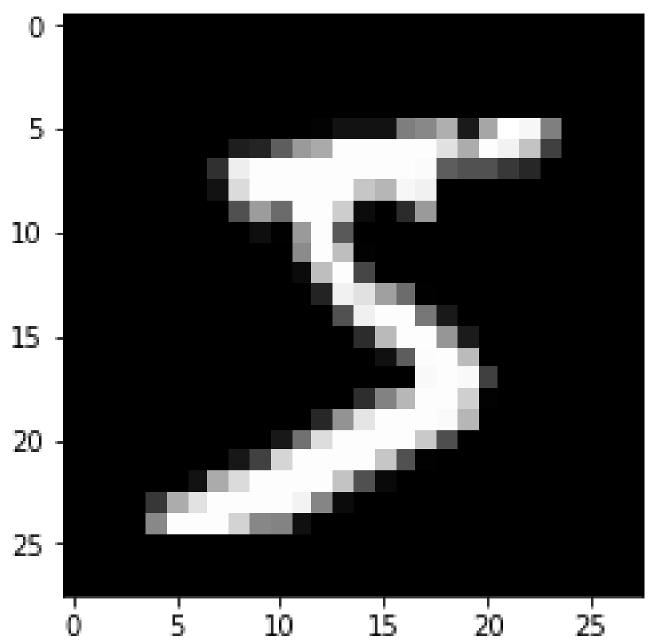
Chainer contains some built-in functions to use some popular datasets like MNIST, CIFAR10/100, etc. Those can automatically download the data from servers and provide dataset objects which are easy to use.

The code below shows how to retrieve the MNIST dataset from the server and save an image from its training split to make sure the images are correctly obtained.

```
from __future__ import print_function
import matplotlib.pyplot as plt
from chainer.datasets import mnist

# Download the MNIST data if you haven't downloaded it yet
train, test = mnist.get_mnist(withlabel=True, ndim=1)

# Display an example from the MNIST dataset.
# `x` contains the input image array and `t` contains that target class
# label as an integer.
x, t = train[0]
plt.imshow(x.reshape(28, 28), cmap='gray')
plt.savefig('5.png')
print('label:', t)
```

```
label: 5
```

The saved image 5.png will look like:



3.2.2 2. Create a dataset iterator

Although this is an optional step, we'd like to introduce the *Iterator* class that retrieves a set of data and labels from the given dataset to easily make a mini-batch. There are some subclasses that can perform the same thing in different ways, e.g., using multi-processing to parallelize the data loading part, etc.

Here, we use *SerialIterator*, which is also a subclass of *Iterator* in the example code below. The *SerialIterator* can provide mini-batches with or without shuffling the order of data in the given dataset.

All *Iterator*s produce a new mini-batch by calling its *next()* method. All *Iterator*s also have properties to know how many times we have taken all the data from the given dataset (epoch) and whether the next mini-batch will be the start of a new epoch (is_new_epoch), and so on.

The code below shows how to create a *SerialIterator* object from a dataset object.

```
from chainer import iterators

# Choose the minibatch size.
batchsize = 128

train_iter = iterators.SerialIterator(train, batchsize)
test_iter = iterators.SerialIterator(test, batchsize,
                                     repeat=False, shuffle=False)
```

Note: *Iterator*s can take a built-in Python list as a given dataset. It means that the example code below is able to work,

```
train = [(x1, t1), (x2, t2), ...]  # A list of tuples
train_iter = iterators.SerialIterator(train, batchsize)
```

where `x1, x2, ...` denote the input data and `t1, t2, ...` denote the corresponding labels.

Details of SerialIterator

- *SerialIterator* is a built-in subclass of *Iterator* that can retrieve a mini-batch from a given dataset in either sequential or shuffled order.
- The *Iterator*'s constructor takes two arguments: a dataset object and a mini-batch size.
- If you want to use the same dataset repeatedly during the training process, set the `repeat` argument to `True` (default). Otherwise, the dataset will be used only one time. The latter case is actually for the evaluation.
- If you want to shuffle the training dataset every epoch, set the `shuffle` argument to `True`. Otherwise, the order of each data retrieved from the dataset will be always the same at each epoch.

In the example code shown above, we set `batchsize = 128` in both `train_iter` and `test_iter`. So, these iterators will provide 128 images and corresponding labels at a time.

3.2.3 3. Define a network

Now let's define a neural network that we will train to classify the MNIST images. For simplicity, we use a three-layer perceptron here. We set each hidden layer to have 100 units and set the output layer to have 10 units, which is corresponding to the number of class labels of the MNIST.

Create your network as a subclass of Chain

You can create your network by writing a new subclass of *Chain*. The main steps are twofold:

1. Register the network components which have trainable parameters to the subclass. Each of them must be instantiated and assigned to a property in the scope specified by *init_scope()*:
2. Define a `forward()` method that represents the actual **forward computation** of your network. This method takes one or more *Variable*, `numpy.ndarray`, or `cupy.ndarray` as its inputs and calculates the forward pass using them.

   ```
   class MyNetwork(Chain):

       def __init__(self, n_mid_units=100, n_out=10):
   ```

(continues on next page)

(continued from previous page)

```
        super(MyNetwork, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_mid_units)
            self.l2 = L.Linear(n_mid_units, n_mid_units)
            self.l3 = L.Linear(n_mid_units, n_out)

    def forward(self, x):
        h = F.relu(self.l1(x))
        h = F.relu(self.l2(h))
        return self.l3(h)

model = MyNetwork()

gpu_id = 0  # Set to -1 if you use CPU
if gpu_id >= 0:
    model.to_gpu(gpu_id)
```

`Link`, `Chain`, `ChainList`, and those subclass objects which contain trainable parameters should be registered to the model by assigning it as a property inside the `init_scope()`. For example, a `FunctionNode` does not contain any trainable parameters, so there is no need to keep the object as a property of your network. When you want to use `relu()` in your network, using it as a function in `forward()` works correctly.

In Chainer, the Python code that implements the forward computation itself represents the network. In other words, we can conceptually think of the computation graph for our network being constructed dynamically as this forward computation code executes. This allows Chainer to describe networks in which different computations can be performed in each iteration, such as branched networks, intuitively and with a high degree of flexibility. This is the key feature of Chainer that we call **Define-by-Run**.

3.2.4 4. Select an optimization algorithm

Chainer provides a wide variety of optimization algorithms that can be used to optimize the network parameters during training. They are located in `optimizers` module.

Here, we are going to use the stochastic gradient descent (SGD) method with momentum, which is implemented by `MomentumSGD`. To use the optimizer, we give the network object (typically it's a `Chain` or `ChainList`) to the `setup()` method of the optimizer object to register it. In this way, the `Optimizer` can automatically find the model parameters and update them during training.

You can easily try out other optimizers as well. Please test and observe the results of various optimizers. For example, you could try to change `MomentumSGD` to `Adam`, `RMSprop`, etc.

```
from chainer import optimizers

# Choose an optimizer algorithm
optimizer = optimizers.MomentumSGD(lr=0.01, momentum=0.9)

# Give the optimizer a reference to the model so that it
# can locate the model's parameters.
optimizer.setup(model)
```

Note: In the above example, we set `lr` to 0.01 in the constructor. This value is known as the "learning rate", one of the most important hyperparameters that need to be adjusted in order to obtain the best performance. The various optimizers may each have different hyperparameters and so be sure to check the documentation for the details.

3.2.5 5. Write a training loop

We now show how to write the training loop. Since we are working on a digit classification problem, we will use `softmax_cross_entropy()` as the loss function for the optimizer to minimize. For other types of problems, such as regression models, other loss functions might be more appropriate. See the Chainer documentation for detailed information on the various loss functions for more details.

Our training loop will be structured as follows.

1. We will first get a mini-batch of examples from the training dataset.
2. We will then feed the batch into our network by calling it (a `Chain` object) like a function. This will execute the forward-pass code that are written in the `forward()` method.
3. This will return the network output that represents class label predictions. We supply it to the loss function along with the true (that is, target) values. The loss function will output the loss as a `Variable` object.
4. We then clear any previous gradients in the network and perform the backward pass by calling the `backward()` method on the loss variable which computes the parameter gradients. We need to clear the gradients first because the `backward()` method accumulates gradients instead of overwriting the previous values.
5. Since the optimizer already has a reference to the network, it has access to the parameters and the computed gradients so that we can now call the `update()` method of the optimizer which will update the model parameters.

In addition to the above steps, you might want to check the performance of the network with a validation dataset. This allows you to observe how well it is generalized to new data so far, namely, you can check whether it is overfitting to the training data. The code below checks the performance on the test set at the end of each epoch. The code has the same structure as the training code except that no backpropagation is performed and we also compute the accuracy on the test data using the `accuracy()` function.

The training loop code is as follows:

```
import numpy as np
from chainer.dataset import concat_examples
from chainer.backends.cuda import to_cpu

max_epoch = 10

while train_iter.epoch < max_epoch:

    # ---------- One iteration of the training loop ----------
    train_batch = train_iter.next()
    image_train, target_train = concat_examples(train_batch, gpu_id)

    # Calculate the prediction of the network
    prediction_train = model(image_train)

    # Calculate the loss with softmax_cross_entropy
    loss = F.softmax_cross_entropy(prediction_train, target_train)

    # Calculate the gradients in the network
    model.cleargrads()
    loss.backward()

    # Update all the trainable parameters
    optimizer.update()
    # --------------------- until here ---------------------
```

(continues on next page)

(continued from previous page)

```
    # Check the validation accuracy of prediction after every epoch
    if train_iter.is_new_epoch:  # If this iteration is the final iteration of the current epoch

        # Display the training loss
        print('epoch:{:02d} train_loss:{:.04f} '.format(
            train_iter.epoch, float(to_cpu(loss.array))), end='')

        test_losses = []
        test_accuracies = []
        for test_batch in test_iter:
            image_test, target_test = concat_examples(test_batch, gpu_id)

            # Forward the test data
            prediction_test = model(image_test)

            # Calculate the loss
            loss_test = F.softmax_cross_entropy(prediction_test, target_test)
            test_losses.append(to_cpu(loss_test.array))

            # Calculate the accuracy
            accuracy = F.accuracy(prediction_test, target_test)
            accuracy.to_cpu()
            test_accuracies.append(accuracy.array)

        test_iter.reset()

        print('val_loss:{:.04f} val_accuracy:{:.04f}'.format(
            np.mean(test_losses), np.mean(test_accuracies)))
```

Output

```
epoch:01 train_loss:0.8072 val_loss:0.7592 val_accuracy:0.8289
epoch:02 train_loss:0.5021 val_loss:0.4467 val_accuracy:0.8841
epoch:03 train_loss:0.3539 val_loss:0.3673 val_accuracy:0.9007
epoch:04 train_loss:0.2524 val_loss:0.3307 val_accuracy:0.9067
epoch:05 train_loss:0.4232 val_loss:0.3076 val_accuracy:0.9136
epoch:06 train_loss:0.3033 val_loss:0.2910 val_accuracy:0.9167
epoch:07 train_loss:0.2004 val_loss:0.2773 val_accuracy:0.9222
epoch:08 train_loss:0.2885 val_loss:0.2679 val_accuracy:0.9239
epoch:09 train_loss:0.2818 val_loss:0.2579 val_accuracy:0.9266
epoch:10 train_loss:0.2403 val_loss:0.2484 val_accuracy:0.9307
```

3.2.6 6. Save the trained model

Chainer provides two types of `serializers` that can be used to save and restore model state. One supports the HDF5 format and the other supports the NumPy NPZ format. For this example, we are going to use the NPZ format to save our model since it is easy to use with NumPy and doesn't need to install any additional dependencies or libraries.

```
serializers.save_npz('my_mnist.model', model)
```

3.2.7 7. Perform classification by the saved model

Let's use the saved model to classify a new image. In order to load the trained model parameters, we need to perform the following two steps:

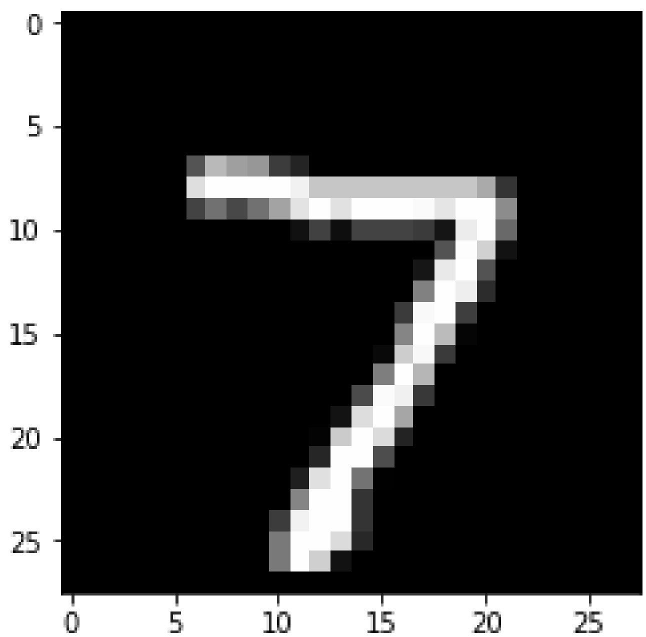
1. Instantiate the same network as what you trained.
2. Overwrite all parameters in the model instance with the saved weights using the `load_npz()` function.

Once the model is restored, it can be used to predict image labels on new input data.

```
from chainer import serializers

# Create an instance of the network you trained
model = MyNetwork()

# Load the saved parameters into the instance
serializers.load_npz('my_mnist.model', model)

# Get a test image and label
x, t = test[0]
plt.imshow(x.reshape(28, 28), cmap='gray')
plt.savefig('7.png')
print('label:', t)
```

```
label: 7
```

The saved test image looks like:




```
# Change the shape of the minibatch.
# In this example, the size of minibatch is 1.
# Inference using any mini-batch size can be performed.

print(x.shape, end=' -> ')
x = x[None, ...]
```

(continues on next page)

(continued from previous page)

```
print(x.shape)

# Forward calculation of the model by sending X
y = model(x)

# The result is given as Variable, then we can take a look at the contents by the␣
↪attribute, .array.
y = y.array

# Look up the most probable digit number using argmax
pred_label = y.argmax(axis=1)

print('predicted label:', pred_label[0])
```

```
(784,) -> (1, 784)
predicted label: 7
```

The prediction result looks correct. Yay!

3.3 Convolutional Network for Visual Recognition Tasks

In this section, you will learn how to write

- A small convolutional network with a model class that is inherited from *Chain*,
- A large convolutional network that has several building block networks with *ChainList*.

After reading this section, you will be able to:

- Write your own original convolutional network in Chainer

A convolutional network (ConvNet) is mainly comprised of convolutional layers. This type of network is commonly used for various visual recognition tasks, e.g., classifying hand-written digits or natural images into given object classes, detecting objects from an image, and labeling all pixels of an image with the object classes (semantic segmentation), and so on.

In such tasks, a typical ConvNet takes a set of images whose shape is (N, C, H, W), where

- N denotes the number of images in a mini-batch,
- C denotes the number of channels of those images,
- H and W denote the height and width of those images,

respectively. Then, it typically outputs a fixed-sized vector as membership probabilities over the target object classes. It also can output a set of feature maps that have the corresponding size to the input image for a pixel labeling task, etc.

> In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
```

(continues on next page)

(continued from previous page)

```
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

3.3.1 LeNet5

Here, let's start by defining LeNet5 [LeCun98] in Chainer. In this example, we show a simplified version of LeNet5 introduced in Deep Learning Tutorials. This is a ConvNet model that has 5 layers comprised of 3 convolutional layers and 2 fully-connected layers. This was proposed to classify hand-written digit images in 1998. In Chainer, the model can be written as follows:

```
class LeNet5(Chain):
    def __init__(self):
        super(LeNet5, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(
                in_channels=1, out_channels=6, ksize=5, stride=1)
            self.conv2 = L.Convolution2D(
                in_channels=6, out_channels=16, ksize=5, stride=1)
            self.conv3 = L.Convolution2D(
                in_channels=16, out_channels=120, ksize=4, stride=1)
            self.fc4 = L.Linear(None, 84)
            self.fc5 = L.Linear(84, 10)

    def forward(self, x):
        h = F.sigmoid(self.conv1(x))
        h = F.max_pooling_2d(h, 2, 2)
        h = F.sigmoid(self.conv2(h))
        h = F.max_pooling_2d(h, 2, 2)
        h = F.sigmoid(self.conv3(h))
        h = F.sigmoid(self.fc4(h))
        if chainer.config.train:
            return self.fc5(h)
        return F.softmax(self.fc5(h))
```

A typical way to write your network is creating a new class inherited from `Chain` class. When defining your model in this way, typically, all the layers which have trainable parameters are registered to the model by assigning the objects of `Link` as an attribute.

The model class is instantiated before the forward and backward computations. To give input images and label vectors simply by calling the model object like a function, `forward()` is usually defined in the model class. This method performs the forward computation of the model. Chainer uses the powerful autograd system for any computational graphs written with `FunctionNode`s and `Link`s (actually a `Link` calls a corresponding `FunctionNode` inside of it), so that you don't need to explicitly write the code for backward computations in the model. Just prepare the data, then give it to the model. The way this works is the resulting output `Variable` from the forward computation has a `backward()` method to perform autograd. In the above model, `forward()` has a `if` statement at the end to switch its behavior by the Chainer's running mode, i.e., training mode or not. Chainer presents the running mode as a global variable `chainer.config.train`. When it's in training mode, `forward()` returns the output value of the last layer as is to compute the loss later on, otherwise it returns a prediction result by calculating `softmax()`.

It is recommended that you use the global configuration `chainer.config.train` to switch the running mode.

If you don't want to write conv1 and the other layers more than once, you can also write the same model like in this way:

```
from functools import partial

class LeNet5(Chain):
    def __init__(self):
        super(LeNet5, self).__init__()
        net = [('conv1', L.Convolution2D(1, 6, 5, 1))]
        net += [('_sigm1', F.sigmoid)]
        net += [('_mpool1', partial(F.max_pooling_2d, ksize=2, stride=2))]
        net += [('conv2', L.Convolution2D(6, 16, 5, 1))]
        net += [('_sigm2', F.sigmoid)]
        net += [('_mpool2', partial(F.max_pooling_2d, ksize=2, stride=2))]
        net += [('conv3', L.Convolution2D(16, 120, 4, 1))]
        net += [('_sigm3', F.sigmoid)]
        net += [('_mpool3', partial(F.max_pooling_2d, ksize=2, stride=2))]
        net += [('fc4', L.Linear(None, 84))]
        net += [('_sigm4', F.sigmoid)]
        net += [('fc5', L.Linear(84, 10))]
        net += [('_sigm5', F.sigmoid)]
        with self.init_scope():
            for n in net:
                if not n[0].startswith('_'):
                    setattr(self, n[0], n[1])
        self.layers = net

    def forward(self, x):
        for n, f in self.layers:
            if not n.startswith('_'):
                x = getattr(self, n)(x)
            else:
                x = f(x)
        if chainer.config.train:
            return x
        return F.softmax(x)
```

Note: You can also use *Sequential* to write the above model more simply. Please note that *Sequential* is an experimental feature introduced in Chainer v4 and its interface may be changed in the future versions.

This code creates a list of pairs of component name (e.g., conv1, _sigm1, etc.) and all *Link*s and functions (e.g., F.sigmoid, which internally invokes *FunctionNode*) after calling its superclass's constructor. In this case, components whose name start with _ are functions (*FunctionNode*), which doesn't have any trainable parameters, so that we don't register (setattr) it to the model. Others (conv1, fc4, etc.) are *Link*s, which are trainable layers that hold parameters. This operation can be freely replaced with many other ways because those component names are just designed to select *Link*s only from the list net easily. The list net is stored as an attribute layers to refer it in forward(). In forward(), it retrieves all layers in the network from self.forward sequentially and gives the input variable or the intermediate output from the previous layer to the current layer. The last part of the forward() to switch its behavior by the training/inference mode is the same as the former way.

Ways to calculate loss

When you train the model with label vector t, the loss should be calculated using the output from the model. There also are several ways to calculate the loss:

```
model = LeNet5()

# Input data and label
x = np.random.rand(32, 1, 28, 28).astype(np.float32)
t = np.random.randint(0, 10, size=(32,)).astype(np.int32)

# Forward computation
y = model(x)

# Loss calculation
loss = F.softmax_cross_entropy(y, t)
```

This is a primitive way to calculate a loss value from the output of the model. On the other hand, the loss computation can be included in the model itself by wrapping the model object (`Chain` or `ChainList` object) with a class inherited from `Chain`. The outer `Chain` should take the model defined above and register it with `init_scope()`. `Chain` is actually inherited from `Link`, so that `Chain` itself can also be registered as a trainable `Link` to another `Chain`. Actually, `Classifier` class to wrap the model and add the loss computation to the model already exists. Actually, there is already a `Classifier` class that can be used to wrap the model and include the loss computation as well. It can be used like this:

```
model = L.Classifier(LeNet5())

# Foward & Loss calculation
loss = model(x, t)
```

This class takes a model object as an input argument and registers it to a `predictor` property as a trained parameter. As shown above, the returned object can then be called like a function in which we pass `x` and `t` as the input arguments and the resulting loss value (which we recall is a `Variable`) is returned.

See the detailed implementation of `Classifier` from here: `chainer.links.Classifier` and check the implementation by looking at the source.

From the above examples, we can see that Chainer provides the flexibility to write our original network in many different ways. Such flexibility intends to make it intuitive for users to design new and complex models.

3.3.2 VGG16

Next, let's write some larger models in Chainer. When you write a large network consisting of several building block networks, `ChainList` is useful. First, let's see how to write a VGG16 [Simonyan14] model.

```
class VGG16(chainer.ChainList):
    def __init__(self):
        super(VGG16, self).__init__(
            VGGBlock(64),
            VGGBlock(128),
            VGGBlock(256, 3),
            VGGBlock(512, 3),
            VGGBlock(512, 3, True))

    def forward(self, x):
        for f in self.children():
            x = f(x)
        if chainer.config.train:
            return x
        return F.softmax(x)

```

(continues on next page)

(continued from previous page)

```
class VGGBlock(chainer.Chain):
    def __init__(self, n_channels, n_convs=2, fc=False):
        w = chainer.initializers.HeNormal()
        super(VGGBlock, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(None, n_channels, 3, 1, 1, initialW=w)
            self.conv2 = L.Convolution2D(
                n_channels, n_channels, 3, 1, 1, initialW=w)
            if n_convs == 3:
                self.conv3 = L.Convolution2D(
                    n_channels, n_channels, 3, 1, 1, initialW=w)
            if fc:
                self.fc4 = L.Linear(None, 4096, initialW=w)
                self.fc5 = L.Linear(4096, 4096, initialW=w)
                self.fc6 = L.Linear(4096, 1000, initialW=w)

        self.n_convs = n_convs
        self.fc = fc

    def forward(self, x):
        h = F.relu(self.conv1(x))
        h = F.relu(self.conv2(h))
        if self.n_convs == 3:
            h = F.relu(self.conv3(h))
        h = F.max_pooling_2d(h, 2, 2)
        if self.fc:
            h = F.dropout(F.relu(self.fc4(h)))
            h = F.dropout(F.relu(self.fc5(h)))
            h = self.fc6(h)
        return h
```

That's it. VGG16 is a model which won the 1st place in classification + localization task at ILSVRC 2014, and since then, has become one of the standard models for many different tasks as a pre-trained model. This has 16-layers, so it's called "VGG-16", but we can write this model without writing all layers independently. Since this model consists of several building blocks that have the same architecture, we can build the whole network by re-using the building block definition. Each part of the network is consisted of 2 or 3 convolutional layers and activation function (`relu()`) following them, and `max_pooling_2d()` operations. This block is written as `VGGBlock` in the above example code. And the whole network just calls this block one by one in sequential manner.

3.3.3 ResNet152

How about ResNet? ResNet [He16] came in the following year's ILSVRC. It is a much deeper model than VGG16, having up to 152 layers. This sounds super laborious to build, but it can be implemented in almost same manner as VGG16. In the other words, it's easy. One possible way to write ResNet-152 is:

```
class ResNet152(chainer.Chain):
    def __init__(self, n_blocks=[3, 8, 36, 3]):
        w = chainer.initializers.HeNormal()
        super(ResNet152, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(None, 64, 7, 2, 3, initialW=w, nobias=True)
            self.bn1 = L.BatchNormalization(64)
            self.res2 = ResBlock(n_blocks[0], 64, 64, 256, 1)
            self.res3 = ResBlock(n_blocks[1], 256, 128, 512)
```

(continues on next page)

(continued from previous page)

```
            self.res4 = ResBlock(n_blocks[2], 512, 256, 1024)
            self.res5 = ResBlock(n_blocks[3], 1024, 512, 2048)
            self.fc6 = L.Linear(2048, 1000)

    def forward(self, x):
        h = self.bn1(self.conv1(x))
        h = F.max_pooling_2d(F.relu(h), 2, 2)
        h = self.res2(h)
        h = self.res3(h)
        h = self.res4(h)
        h = self.res5(h)
        h = F.average_pooling_2d(h, h.shape[2:], stride=1)
        h = self.fc6(h)
        if chainer.config.train:
            return h
        return F.softmax(h)


class ResBlock(chainer.ChainList):
    def __init__(self, n_layers, n_in, n_mid, n_out, stride=2):
        super(ResBlock, self).__init__()
        self.add_link(BottleNeck(n_in, n_mid, n_out, stride, True))
        for _ in range(n_layers - 1):
            self.add_link(BottleNeck(n_out, n_mid, n_out))

    def forward(self, x):
        for f in self.children():
            x = f(x)
        return x


class BottleNeck(chainer.Chain):
    def __init__(self, n_in, n_mid, n_out, stride=1, proj=False):
        w = chainer.initializers.HeNormal()
        super(BottleNeck, self).__init__()
        with self.init_scope():
            self.conv1x1a = L.Convolution2D(
                n_in, n_mid, 1, stride, 0, initialW=w, nobias=True)
            self.conv3x3b = L.Convolution2D(
                n_mid, n_mid, 3, 1, 1, initialW=w, nobias=True)
            self.conv1x1c = L.Convolution2D(
                n_mid, n_out, 1, 1, 0, initialW=w, nobias=True)
            self.bn_a = L.BatchNormalization(n_mid)
            self.bn_b = L.BatchNormalization(n_mid)
            self.bn_c = L.BatchNormalization(n_out)
            if proj:
                self.conv1x1r = L.Convolution2D(
                    n_in, n_out, 1, stride, 0, initialW=w, nobias=True)
                self.bn_r = L.BatchNormalization(n_out)
        self.proj = proj

    def forward(self, x):
        h = F.relu(self.bn_a(self.conv1x1a(x)))
        h = F.relu(self.bn_b(self.conv3x3b(h)))
        h = self.bn_c(self.conv1x1c(h))
        if self.proj:
            x = self.bn_r(self.conv1x1r(x))
```

(continues on next page)

(continued from previous page)

```
        return F.relu(h + x)
```

In the `BottleNeck` class, depending on the value of the proj argument supplied to the initializer, it will conditionally compute a convolutional layer `conv1x1r` which will extend the number of channels of the input `x` to be equal to the number of channels of the output of `conv1x1c`, and followed by a batch normalization layer before the final ReLU layer. Writing the building block in this way improves the re-usability of a class. It switches not only the behavior in `__class__()` by flags but also the parameter registration. In this case, when `proj` is `False`, the `BottleNeck` doesn't have *conv1x1r* and *bn_r* layers, so the memory usage would be efficient compared to the case when it registers both anyway and just ignore them if `proj` is `False`.

Using nested *Chain*s and *ChainList* for sequential part enables us to write complex and very deep models easily.

3.3.4 Use Pre-trained Models

Various ways to write your models were described above. It turns out that VGG16 and ResNet are very useful as general feature extractors for many kinds of tasks, including but not limited to image classification. So, Chainer provides you with the pre-trained VGG16 and ResNet-50/101/152 models with a simple API. You can use these models as follows:

```
from chainer.links import VGG16Layers

model = VGG16Layers()
```

When *VGG16Layers* is instantiated, the pre-trained parameters are automatically downloaded from the author's server. So you can immediately start to use VGG16 with pre-trained weight as a good image feature extractor. See the details of this model here: *chainer.links.VGG16Layers*.

In the case of ResNet models, there are three variations differing in the number of layers. We have *chainer.links.ResNet50Layers*, *chainer.links.ResNet101Layers*, and *chainer.links.ResNet152Layers* models with easy parameter loading feature. ResNet's pre-trained parameters are not available for direct downloading, so you need to download the weight from the author's web page first, and then place it into the dir `$CHAINER_DATSET_ROOT/pfnet/chainer/models` or your favorite place. Once the preparation is finished, the usage is the same as VGG16:

```
from chainer.links import ResNet152Layers

model = ResNet152Layers()
```

```
Traceback (most recent call last):
OSError: The pre-trained caffemodel does not exist. Please download it from 'https://
↪github.com/KaimingHe/deep-residual-networks', and place it on ...
```

Please see the details of usage and how to prepare the pre-trained weights for ResNet here: *chainer.links.ResNet50Layers*

References

3.4 DCGAN: Generate images with Deep Convolutional GAN

3.4.1 0. Introduction

In this tutorial, we generate images with **generative adversarial networks (GAN)**. GAN are kinds of deep neural network for generative modeling that are often applied to image generation. GAN-based models are also used in PaintsChainer, an automatic colorization service.

In this tutorial, you will learn the following things:

1. Generative Adversarial Networks (GAN)
2. Implementation of DCGAN in Chainer

3.4.2 1. Generarive Adversarial Networks (GAN)

1.1 What are GAN?

As explained in GAN tutorial in NIPS 2016 [1], generative models can be classified into the categories as shown in the following figure:

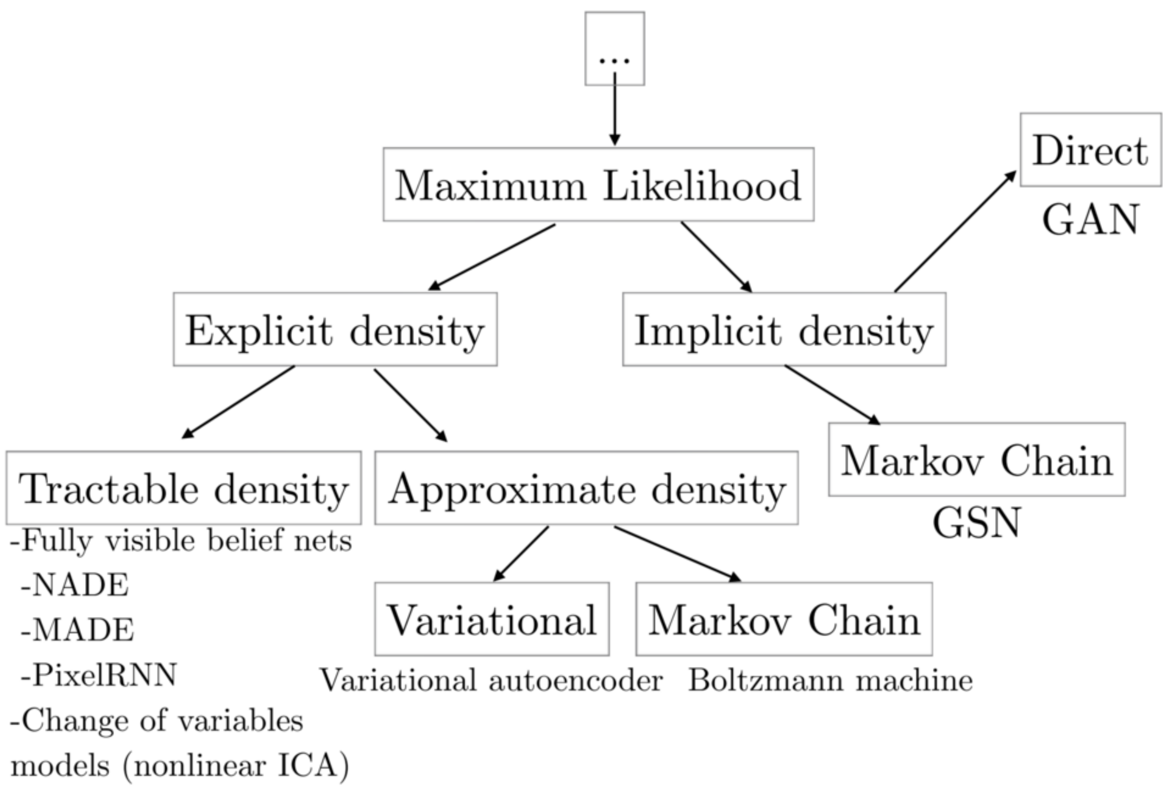


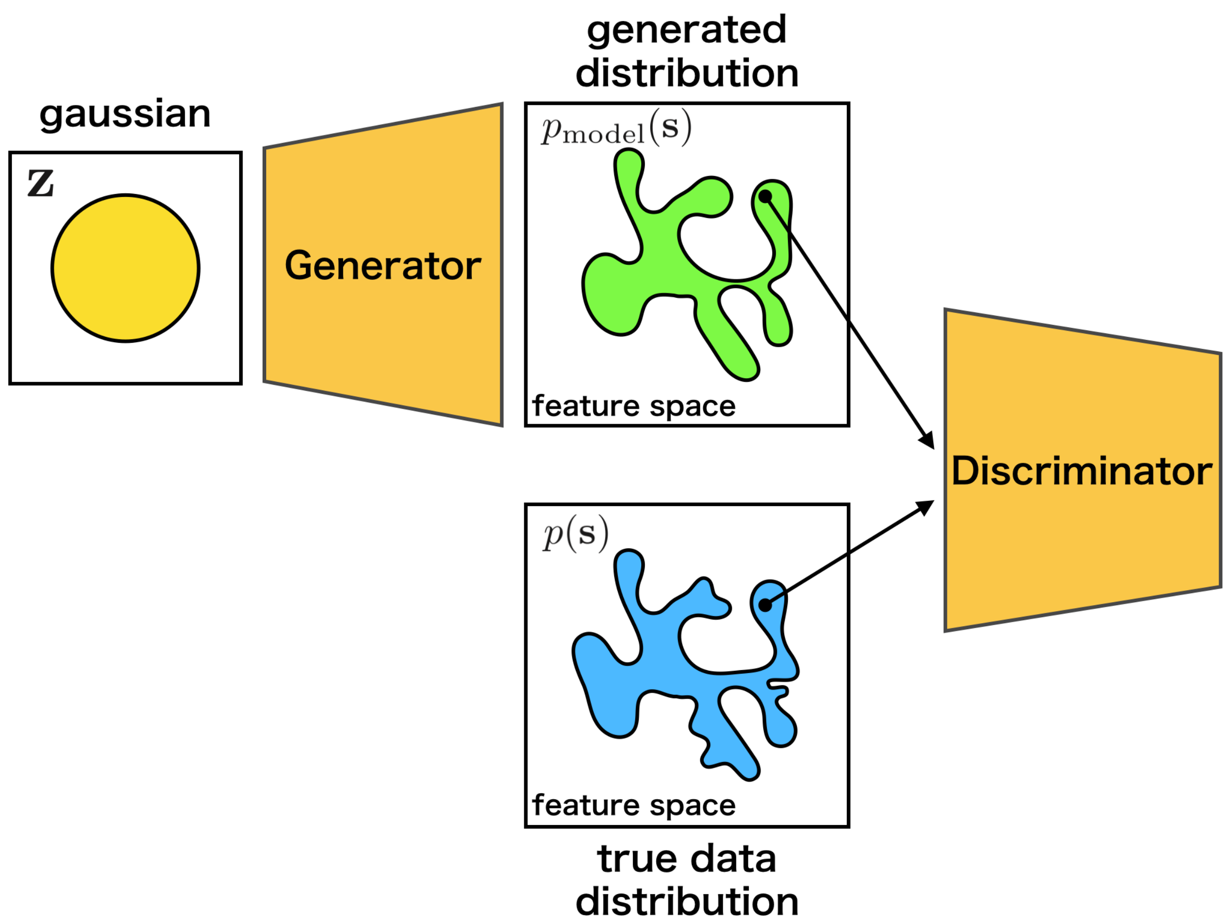
Fig. 1: cited from [1]

Besides GAN, other famous generative models include Fully visible belief networks (FVBNs) and Variational autoencoder (VAE). Unlike FVBNs and VAE, GAN do not explicitly model the probability distribution $p(\mathbf{s})$ that generates training data. Instead, we model a generator $G : \mathbf{z} \mapsto \mathbf{s}$. The generator G samples $\mathbf{s} \sim p(\mathbf{s})$ from the latent variable $\mathbf{z}$. Apart from the generator G, we create a discriminator $D(\mathbf{x})$ which discriminates between samples from the generator G and examples from training data. While training the discriminator D, the generator G tries to maximize the probability of the discriminator D making a mistake. So, the generator G tries to create samples that seem to be drawn from the same distribution as the training data.

The advantages of GAN are low sampling cost and its state-of-the-art performance in image generation. The disadvantage is that we cannot calculate the likelihood $p_{\mathrm{model}}(\mathbf{s})$ because we do not model any probability distribution, and we cannot infer the latent variable $\mathbf{z}$ from a sample.

1.2 How GAN work?

As explained above, GAN use the two models, the generator and the discriminator. When training the networks, we should match the data distribution $p(\mathbf{s})$ with the distribution of the samples $\mathbf{s} = G(\mathbf{z})$ generated from the generator.



The generator G learns the target distribution, and ideally eventually reaches a **Nash equilibrium** [2] of game theory. In detail, while training the discriminator D, the generator G is also trained, so that the discriminator D makes a mistake.

As an intuitive example, the relationship between counterfeiters of banknotes and the police is frequently used. The counterfeiters try to make counterfeit notes that look like real banknotes. The police try to distinguish real bank notes from counterfeit notes. It is supposed that the ability of the police gradually rises, so that real banknotes and counterfeit notes can be recognized well. Then, the counterfeiters will not be able to use counterfeit banknotes, so they will create counterfeit banknotes that appear more realistic. As the police improve their skill further, they can distinguish real and counterfeit notes… Eventually, the counterfeiter will be able to produce counterfeit banknotes look as real as genuine ones.

The training process is explained by the following mathematical expressions. First, since the discriminator $D(\mathbf{s})$ is the probability that a sample $\mathbf{s}$ is generated from the data distribution at, it can be expressed as follows:

$$D(\mathbf{s}) = \frac{p(\mathbf{s})}{p(\mathbf{s}) + p_{\mathrm{model}}(\mathbf{s})}$$

Then, when we match the data distribution $\mathbf{s} \sim p(\mathbf{s})$ and the distribution of generated samples by G, it means that we should minimize the dissimilarity between the two distributions. It is common to use **Jensen-Shannon Divergence** D_{JS} to measure the dissimilarity between distributions[3].

The D_{JS} of $p_{\mathrm{model}}(\mathbf{s})$ and $p(\mathbf{s})$ can be written as follows by using $D(\mathbf{s})$:

$$\begin{aligned} 2D_{\mathrm{JS}} &= D_{\mathrm{KL}}(p(\mathbf{s})||\bar{p}(\mathbf{s})) + D_{\mathrm{KL}}(p_{\mathrm{model}}(\mathbf{s})||\bar{p}(\mathbf{s})) \\ &= \mathbb{E}_{p(\mathbf{s})}\left[\log \frac{2p(\mathbf{s})}{p(\mathbf{s}) + p_{\mathrm{model}}(\mathbf{s})}\right] + \mathbb{E}_{p_{\mathrm{model}}}\left[\log \frac{2p_{\mathrm{model}}(\mathbf{s})}{p(\mathbf{s}) + p_{\mathrm{model}}(\mathbf{s})}\right] \\ &= \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\mathrm{model}}} \log(1 - D(\mathbf{s})) + \log 4 \\ &= \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\mathbf{z}}} \log(1 - D(G(\mathbf{z}))) + \log 4 \end{aligned}$$

where $\bar{p}(\mathbf{s}) = \frac{p(\mathbf{s}) + p_{\mathrm{model}}(\mathbf{s})}{2}$. The D_{JS} will be maximized by the discriminator D and minimized by the generator G, namely, p_{model}. And the distribution $p_{model}(\mathbf{s})$ generated by $G(\mathbf{s})$ can match the data distribution $p(\mathbf{s})$.

$$\min_G \max_D \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\mathbf{z}}} \log(1 - D(G(\mathbf{z})))$$

When we actually train the model, the above min-max problem is solved by alternately updating the discriminator $D(\mathbf{s})$ and the generator $G(\mathbf{z})$ [4]. The actual training procedures are described as follows:

1.3 What are DCGAN?

In this section, we will introduce the model called DCGAN(Deep Convolutional GAN) proposed by Radford et al.[5]. As shown below, it is a model using CNN(Convolutional Neural Network) as its name suggests.

In addition, although GAN are known for its difficulty in training, this paper introduces various techniques for successful training:

1. Convert max-pooling layers to convolution layers with larger or fractional strides
2. Convert fully connected layers to global average pooling layers in the discriminator
3. Use batch normalization layers in the generator and the discriminator
4. Use leaky ReLU activation functions in the discriminator

3.4.3 2. Implementation of DCGAN in Chainer

There is an example of DCGAN in the official repository of Chainer, so we will explain how to implement DCGAN based on this: chainer/examples/dcgan

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Fig. 2: cited from [4]

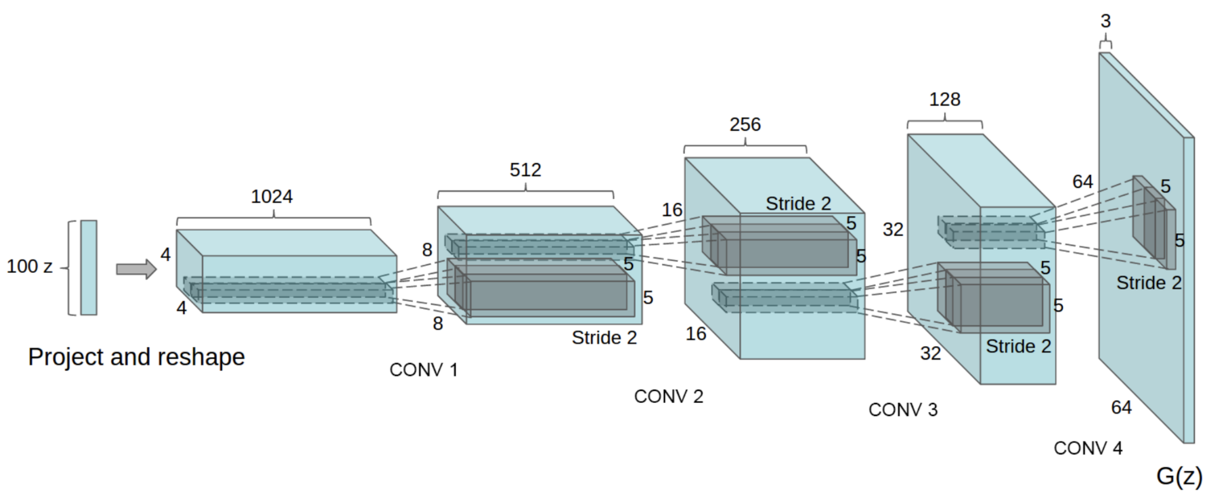


Fig. 3: cited from [5]

2.1 Define the generator model

First, let's define a network for the generator.

Listing 1: train_dcgan.py

```
class Generator(chainer.Chain):

    def __init__(self, n_hidden, bottom_width=4, ch=512, wscale=0.02):
        super(Generator, self).__init__()
        self.n_hidden = n_hidden
        self.ch = ch
        self.bottom_width = bottom_width

        with self.init_scope():
            w = chainer.initializers.Normal(wscale)
            self.l0 = L.Linear(self.n_hidden, bottom_width * bottom_width * ch,
                               initialW=w)
            self.dc1 = L.Deconvolution2D(ch, ch // 2, 4, 2, 1, initialW=w)
            self.dc2 = L.Deconvolution2D(ch // 2, ch // 4, 4, 2, 1, initialW=w)
            self.dc3 = L.Deconvolution2D(ch // 4, ch // 8, 4, 2, 1, initialW=w)
            self.dc4 = L.Deconvolution2D(ch // 8, 3, 3, 1, 1, initialW=w)
            self.bn0 = L.BatchNormalization(bottom_width * bottom_width * ch)
            self.bn1 = L.BatchNormalization(ch // 2)
            self.bn2 = L.BatchNormalization(ch // 4)
            self.bn3 = L.BatchNormalization(ch // 8)

    def make_hidden(self, batchsize):
        dtype = chainer.get_dtype()
        return numpy.random.uniform(-1, 1, (batchsize, self.n_hidden, 1, 1))\
            .astype(dtype)

    def forward(self, z):
        h = F.reshape(F.relu(self.bn0(self.l0(z))),
                      (len(z), self.ch, self.bottom_width, self.bottom_width))
        h = F.relu(self.bn1(self.dc1(h)))
        h = F.relu(self.bn2(self.dc2(h)))
        h = F.relu(self.bn3(self.dc3(h)))
        x = F.sigmoid(self.dc4(h))
        return x
```

When we make a network in Chainer, there are some conventions:

1. Define a network class which inherits `Chain`.
2. Make `chainer.links`'s instances in the `init_scope():` of the initializer `__init__`.
3. Define network connections in the `__call__` operator by using the `chainer.links`'s instances and `chainer.functions`.

If you are not familiar with constructing a new network, please refer to *this tutorial*.

As we can see from the initializer `__init__`, the `Generator` uses deconvolution layers `Deconvolution2D` and batch normalization layers `BatchNormalization`. In `__call__`, each layer is called and followed by `relu` except the last layer.

Because the first argument of `L.Deconvolution` is the channel size of input and the second is the channel size of output, we can find that each layer halves the channel size. When we construct `Generator` with `ch=1024`, the network is same as the above image.

Note: Be careful when passing the output of a fully connected layer to a convolution layer, because the convolutional layer needs additional dimensions for inputs. As we can see the 1st line of `__call__`, the output of the fully connected layer is reshaped by *reshape* to add the dimensions of the channel, the width and the height of images.

2.2 Define the discriminator model

In addtion, let's define the network for the discriminator.

Listing 2: train_dcgan.py

```
class Discriminator(chainer.Chain):

    def __init__(self, bottom_width=4, ch=512, wscale=0.02):
        w = chainer.initializers.Normal(wscale)
        super(Discriminator, self).__init__()
        with self.init_scope():
            self.c0_0 = L.Convolution2D(3, ch // 8, 3, 1, 1, initialW=w)
            self.c0_1 = L.Convolution2D(ch // 8, ch // 4, 4, 2, 1, initialW=w)
            self.c1_0 = L.Convolution2D(ch // 4, ch // 4, 3, 1, 1, initialW=w)
            self.c1_1 = L.Convolution2D(ch // 4, ch // 2, 4, 2, 1, initialW=w)
            self.c2_0 = L.Convolution2D(ch // 2, ch // 2, 3, 1, 1, initialW=w)
            self.c2_1 = L.Convolution2D(ch // 2, ch // 1, 4, 2, 1, initialW=w)
            self.c3_0 = L.Convolution2D(ch // 1, ch // 1, 3, 1, 1, initialW=w)
            self.l4 = L.Linear(bottom_width * bottom_width * ch, 1, initialW=w)
            self.bn0_1 = L.BatchNormalization(ch // 4, use_gamma=False)
            self.bn1_0 = L.BatchNormalization(ch // 4, use_gamma=False)
            self.bn1_1 = L.BatchNormalization(ch // 2, use_gamma=False)
            self.bn2_0 = L.BatchNormalization(ch // 2, use_gamma=False)
            self.bn2_1 = L.BatchNormalization(ch // 1, use_gamma=False)
            self.bn3_0 = L.BatchNormalization(ch // 1, use_gamma=False)

    def forward(self, x):
        device = self.device
        h = add_noise(device, x)
        h = F.leaky_relu(add_noise(device, self.c0_0(h)))
        h = F.leaky_relu(add_noise(device, self.bn0_1(self.c0_1(h))))
        h = F.leaky_relu(add_noise(device, self.bn1_0(self.c1_0(h))))
        h = F.leaky_relu(add_noise(device, self.bn1_1(self.c1_1(h))))
        h = F.leaky_relu(add_noise(device, self.bn2_0(self.c2_0(h))))
        h = F.leaky_relu(add_noise(device, self.bn2_1(self.c2_1(h))))
        h = F.leaky_relu(add_noise(device, self.bn3_0(self.c3_0(h))))
        return self.l4(h)
```

The `Discriminator` network is almost mirrors of the `Generator` network. However, there are minor different points:

1. Use *leaky_relu* as activation functions
2. Deeper than `Generator`
3. Add some noise to every intermediate outputs before giving them to the next layers

Listing 3: train_dcgan.py

```
def add_noise(device, h, sigma=0.2):
    if chainer.config.train:
```

(continues on next page)

(continued from previous page)

```
        xp = device.xp
        # TODO(niboshi): Support random.randn in ChainerX
        if device.xp is chainerx:
            fallback_device = device.fallback_device
            with chainer.using_device(fallback_device):
                randn = device.send(fallback_device.xp.random.randn(*h.shape))
        else:
            randn = xp.random.randn(*h.shape)
        return h + sigma * randn
    else:
        return h
```

2.3 Prepare dataset and iterator

Let's retrieve the CIFAR-10 dataset by using Chainer's dataset utility function *get_cifar10*. CIFAR-10 is a set of small natural images. Each example is an RGB color image of size 32x32. In the original images, each of R, G, B of pixels is represented by one-byte unsigned integer (i.e. from 0 to 255). This function changes the scale of pixel values into `[0, scale]` float values.

```
    train, _ = chainer.datasets.get_cifar10(withlabel=False, scale=255.)
```

Listing 4: train_dcgan.py

```
train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
```

2.4 Prepare model and optimizer

Let's make the instances of the generator and the discriminator.

Listing 5: train_dcgan.py

```
gen = Generator(n_hidden=args.n_hidden)
dis = Discriminator()

gen.to_device(device)  # Copy the model to the device
dis.to_device(device)

# Setup an optimizer
def make_optimizer(model, alpha=0.0002, beta1=0.5):
    optimizer = chainer.optimizers.Adam(alpha=alpha, beta1=beta1)
    optimizer.setup(model)
    optimizer.add_hook(
        chainer.optimizer_hooks.WeightDecay(0.0001), 'hook_dec')
    return optimizer

opt_gen = make_optimizer(gen)
opt_dis = make_optimizer(dis)
```

Next, let's make optimizers for the models created above.

Listing 6: train_dcgan.py

```
def make_optimizer(model, alpha=0.0002, beta1=0.5):
    optimizer = chainer.optimizers.Adam(alpha=alpha, beta1=beta1)
    optimizer.setup(model)
    optimizer.add_hook(
        chainer.optimizer_hooks.WeightDecay(0.0001), 'hook_dec')
    return optimizer

opt_gen = make_optimizer(gen)
opt_dis = make_optimizer(dis)
```

2.5 Prepare updater

GAN need the two models: the generator and the discriminator. Usually, the default updaters pre-defined in Chainer take only one model. So, we need to define a custom updater for GAN training.

The definition of `DCGANUpdater` is a little complicated. However, it just minimizes the loss of the discriminator and that of the generator alternately.

As you can see in the class definition, `DCGANUpdater` inherits *StandardUpdater*. In this case, almost all necessary functions are defined in *StandardUpdater*, we just override the functions of `__init__` and `update_core`.

Note: We do not need to define `loss_dis` and `loss_gen` because the functions are called only in `update_core`. It aims at improving readability.

Listing 7: train_dcgan.py

```
class DCGANUpdater(chainer.training.updaters.StandardUpdater):

    def __init__(self, *args, **kwargs):
        self.gen, self.dis = kwargs.pop('models')
        super(DCGANUpdater, self).__init__(*args, **kwargs)

    def loss_dis(self, dis, y_fake, y_real):
        batchsize = len(y_fake)
        L1 = F.sum(F.softplus(-y_real)) / batchsize
        L2 = F.sum(F.softplus(y_fake)) / batchsize
        loss = L1 + L2
        chainer.report({'loss': loss}, dis)
        return loss

    def loss_gen(self, gen, y_fake):
        batchsize = len(y_fake)
        loss = F.sum(F.softplus(-y_fake)) / batchsize
        chainer.report({'loss': loss}, gen)
        return loss

    def update_core(self):
        gen_optimizer = self.get_optimizer('gen')
        dis_optimizer = self.get_optimizer('dis')
```

(continues on next page)

(continued from previous page)

```
        batch = self.get_iterator('main').next()
        device = self.device
        x_real = Variable(self.converter(batch, device)) / 255.

        gen, dis = self.gen, self.dis
        batchsize = len(batch)

        y_real = dis(x_real)

        z = Variable(device.xp.asarray(gen.make_hidden(batchsize)))
        x_fake = gen(z)
        y_fake = dis(x_fake)

        dis_optimizer.update(self.loss_dis, dis, y_fake, y_real)
        gen_optimizer.update(self.loss_gen, gen, y_fake)
```

In the initializer `__init__`, an additional keyword argument `models` is required as you can see the code below. Also, we use keyword arguments `iterator`, `optimizer` and `device`. It should be noted that the `optimizer` augment takes a dictionary. The two different models require two different optimizers. To specify the different optimizers for the models, we give a dictionary, `{'gen': opt_gen, 'dis': opt_dis}`, to the `optimizer` argument. we should input `optimizer` as a dictionary `{'gen': opt_gen, 'dis': opt_dis}`. In the `DCGANUpdater`, you can access the iterator with `self.get_iterator('main')`. Also, you can access the optimizers with `self.get_optimizer('gen')` and `self.get_optimizer('dis')`.

In `update_core`, the two loss functions `loss_dis` and `loss_gen` are minimized by the optimizers. At first two lines, we access the optimizers. Then, we create next minibatch of training data by `self.get_iterator('main').next()`, copy `batch` to the device by `self.converter`, and make it a `Variable` object. After that, we minimize the loss functions with the optimizers.

Note: When defining `update_core`, we may want to manipulate the underlying `array` of a `Variable` with `numpy` or `cupy` library. Note that the type of arrays on CPU is `numpy.ndarray`, while the type of arrays on GPU is `cupy.ndarray`. However, users do not need to write `if` condition explicitly, because the appropriate array module can be obtained by `xp = chainer.backend.get_array_module(variable.array)`. If `variable` is on GPU, `cupy` is assigned to `xp`, otherwise `numpy` is assigned to `xp`.

Listing 8: train_dcgan.py

```
updater = DCGANUpdater(
    models=(gen, dis),
    iterator=train_iter,
    optimizer={
        'gen': opt_gen, 'dis': opt_dis},
    device=device)
```

2.6 Prepare trainer and run

Listing 9: train_dcgan.py

```
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)

snapshot_interval = (args.snapshot_interval, 'iteration')
```

(continues on next page)

(continued from previous page)

```
display_interval = (args.display_interval, 'iteration')
trainer.extend(
    extensions.snapshot(filename='snapshot_iter_{.updater.iteration}.npz'),
    trigger=snapshot_interval)
trainer.extend(extensions.snapshot_object(
    gen, 'gen_iter_{.updater.iteration}.npz'), trigger=snapshot_interval)
trainer.extend(extensions.snapshot_object(
    dis, 'dis_iter_{.updater.iteration}.npz'), trigger=snapshot_interval)
trainer.extend(extensions.LogReport(trigger=display_interval))
trainer.extend(extensions.PrintReport([
    'epoch', 'iteration', 'gen/loss', 'dis/loss',
]), trigger=display_interval)
trainer.extend(extensions.ProgressBar(update_interval=10))
trainer.extend(
    out_generated_image(
        gen, dis,
        10, 10, args.seed, args.out),
    trigger=snapshot_interval)
```

Listing 10: train_dcgan.py

```
trainer.run()
```

2.7 Start training

We can run the example as follows.

```
$ pwd
/root2chainer/chainer/examples/dcgan
$ python train_dcgan.py --gpu 0
GPU: 0
# Minibatch-size: 50
# n_hidden: 100
# epoch: 1000

epoch       iteration   gen/loss    dis/loss  ................]  0.01%
0           100         1.2292      1.76914
     total [..................................................]  0.02%
this epoch [#########.........................................] 19.00%
       190 iter, 0 epoch / 1000 epochs
    10.121 iters/sec. Estimated time to finish: 1 day, 3:26:26.372445.
```

The results will be saved in the directory `/root2chainer/chainer/examples/dcgan/result/`. The image is generated by the generator trained for 1000 epochs, and the GIF image on the top of this page shows generated images after every 10 epochs.

3.4.4 3. Reference

- [1] NIPS 2016 Tutorial: Generative Adversarial Networks
- [2] Nash equilibrium
- [3] Jensen-Shannon Divergence
- [4] Generative Adversarial Networks
- [5] Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks

3.5 Recurrent Nets and their Computational Graph

In the example code of this tutorial, we assume for simplicity that the following symbols are already imported.

```
import math
import numpy as np
import chainer
from chainer import backend
from chainer import backends
from chainer.backends import cuda
from chainer import Function, FunctionNode, gradient_check, report, training, utils,␣
↪Variable
from chainer import datasets, initializers, iterators, optimizers, serializers
```

(continues on next page)

(continued from previous page)

```
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

In this section, you will learn how to write

- recurrent nets with full backprop,
- recurrent nets with truncated backprop,
- evaluation of networks with few memory.

After reading this section, you will be able to:

- Handle input sequences of variable length
- Truncate upper stream of the network during forward computation
- Use no-backprop mode to prevent network construction

3.5.1 Recurrent Nets

Recurrent nets are neural networks with loops. They are often used to learn from sequential input/output. Given an input stream $x_1, x_2, \dots, x_t, \dots$ and the initial state h_0, a recurrent net iteratively updates its state by $h_t = f(x_t, h_{t-1})$, and at some or every point in time t, it outputs $y_t = g(h_t)$. If we expand the procedure along the time axis, it looks like a regular feed-forward network except that same parameters are repeatedly used within the network.

Here we learn how to write a simple one-layer recurrent net. The task is language modeling: given a finite sequence of words, we want to predict the next word at each position without peeking the successive words. Suppose there are 1,000 different word types, and that we use 100 dimensional real vectors to represent each word (a.k.a. word embedding).

Let's start from defining the recurrent neural net language model (RNNLM) as a chain. We can use the *chainer.links.LSTM* link that implements a fully-connected stateful LSTM layer. This link looks like an ordinary fully-connected layer. On construction, you pass the input and output size to the constructor:

```
>>> l = L.LSTM(100, 50)
```

Then, call on this instance `l(x)` executes *one step of LSTM layer*:

```
>>> l.reset_state()
>>> x = Variable(np.random.randn(10, 100).astype(np.float32))
>>> y = l(x)
```

Do not forget to reset the internal state of the LSTM layer before the forward computation! Every recurrent layer holds its internal state (i.e. the output of the previous call). At the first application of the recurrent layer, you must reset the internal state. Then, the next input can be directly fed to the LSTM instance:

```
>>> x2 = Variable(np.random.randn(10, 100).astype(np.float32))
>>> y2 = l(x2)
```

Based on this LSTM link, let's write our recurrent network as a new chain:

```
class RNN(Chain):
    def __init__(self):
        super(RNN, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.embed = L.EmbedID(1000, 100)  # word embedding
            self.mid = L.LSTM(100, 50)  # the first LSTM layer
            self.out = L.Linear(50, 1000)  # the feed-forward output layer

    def reset_state(self):
        self.mid.reset_state()

    def forward(self, cur_word):
        # Given the current word ID, predict the next word.
        x = self.embed(cur_word)
        h = self.mid(x)
        y = self.out(h)
        return y

rnn = RNN()
model = L.Classifier(rnn)
optimizer = optimizers.SGD()
optimizer.setup(model)
```

Here `EmbedID` is a link for word embedding. It converts input integers into corresponding fixed-dimensional embedding vectors. The last linear link `out` represents the feed-forward output layer.

The `RNN` chain implements a *one-step-forward computation*. It does not handle sequences by itself, but we can use it to process sequences by just feeding items in a sequence straight to the chain.

Suppose we have a list of word variables `x_list`. Then, we can compute loss values for the word sequence by simple `for` loop.

```
def compute_loss(x_list):
    loss = 0
    for cur_word, next_word in zip(x_list, x_list[1:]):
        loss += model(cur_word, next_word)
    return loss
```

Of course, the accumulated loss is a Variable object with the full history of computation. So we can just call its `backward()` method to compute gradients of the total loss according to the model parameters:

```
# Suppose we have a list of word variables x_list.
rnn.reset_state()
model.cleargrads()
loss = compute_loss(x_list)
loss.backward()
optimizer.update()
```

Or equivalently we can use the `compute_loss` as a loss function:

```
rnn.reset_state()
optimizer.update(compute_loss, x_list)
```

3.5.2 Truncate the Graph by Unchaining

Learning from very long sequences is also a typical use case of recurrent nets. Suppose the input and state sequence is too long to fit into memory. In such cases, we often truncate the backpropagation into a short time range. This technique is called *truncated backprop*. It is heuristic, and it makes the gradients biased. However, this technique works well in practice if the time range is long enough.

How to implement truncated backprop in Chainer? Chainer has a smart mechanism to achieve truncation, called **backward unchaining**. It is implemented in the *Variable.unchain_backward()* method. Backward unchaining starts from the Variable object, and it chops the computation history backwards from the variable. The chopped variables are disposed automatically (if they are not referenced explicitly from any other user object). As a result, they are no longer a part of computation history, and are not involved in backprop anymore.

Let's write an example of truncated backprop. Here we use the same network as the one used in the previous subsection. Suppose we are given a very long sequence, and we want to run backprop truncated at every 30 time steps. We can write truncated backprop using the model defined above:

```
loss = 0
count = 0
seqlen = len(x_list[1:])

rnn.reset_state()
for cur_word, next_word in zip(x_list, x_list[1:]):
    loss += model(cur_word, next_word)
    count += 1
    if count % 30 == 0 or count == seqlen:
        model.cleargrads()
        loss.backward()
        loss.unchain_backward()
        optimizer.update()
```

State is updated at `model()`, and the losses are accumulated to `loss` variable. At each 30 steps, backprop takes place at the accumulated loss. Then, the *unchain_backward()* method is called, which deletes the computation history backward from the accumulated loss. Note that the last state of `model` is not lost, since the RNN instance holds a reference to it.

The implementation of truncated backprop is simple, and since there is no complicated trick on it, we can generalize this method to different situations. For example, we can easily extend the above code to use different schedules between backprop timing and truncation length.

3.5.3 Network Evaluation without Storing the Computation History

On evaluation of recurrent nets, there is typically no need to store the computation history. While unchaining enables us to walk through unlimited length of sequences with limited memory, it is a bit of a work-around.

As an alternative, Chainer provides an evaluation mode of forward computation which does not store the computation history. This is enabled by just calling *no_backprop_mode()* context:

```
with chainer.no_backprop_mode():
    x_list = [Variable(...) for _ in range(100)]  # list of 100 words
    loss = compute_loss(x_list)
```

Note that we cannot call `loss.backward()` to compute the gradient here, since the variable created in the no-backprop context does not remember the computation history.

No-backprop context is also useful to evaluate feed-forward networks to reduce the memory footprint.

We can combine a fixed feature extractor network and a trainable predictor network using *no_backprop_mode()*. For example, suppose we want to train a feed-forward network `predictor_func`, which is located on top of another fixed pre-trained network `fixed_func`. We want to train `predictor_func` without storing the computation history for `fixed_func`. This is simply done by following code snippets (suppose `x_data` and `y_data` indicate input data and label, respectively):

```
with chainer.no_backprop_mode():
    x = Variable(x_data)
    feat = fixed_func(x)
y = predictor_func(feat)
y.backward()
```

At first, the input variable `x` is in no-backprop mode, so `fixed_func` does not memorize the computation history. Then `predictor_func` is executed in backprop mode, i.e., with memorizing the history of computation. Since the history of computation is only memorized between variables `feat` and `y`, the backward computation stops at the `feat` variable.

3.5.4 Making it with Trainer

The above codes are written with plain Function/Variable APIs. When we write a training loop, it is better to use Trainer, since we can then easily add functionalities by extensions.

Before implementing it on Trainer, let's clarify the training settings. We here use Penn Tree Bank dataset as a set of sentences. Each sentence is represented as a word sequence. We concatenate all sentences into one long word sequence, in which each sentence is separated by a special word `<eos>`, which stands for "End of Sequence". This dataset is easily obtained by *`chainer.datasets.get_ptb_words()`*. This function returns train, validation, and test dataset, each of which is represented as a long array of integers. Each integer represents a word ID.

Our task is to learn a recurrent neural net language model from the long word sequence. We use words in different locations to form mini-batches. It means we maintain B indices pointing to different locations in the sequence, read from these indices at each iteration, and increment all indices after the read. Of course, when one index reaches the end of the whole sequence, we turn the index back to 0.

In order to implement this training procedure, we have to customize the following components of Trainer:

- Iterator. Built-in iterators do not support reading from different locations and aggregating them into a mini-batch.
- Update function. The default update function does not support truncated BPTT.

When we write a dataset iterator dedicated to the dataset, the dataset implementation can be arbitrary; even the interface is not fixed. On the other hand, the iterator must support the *`Iterator`* interface. The important methods and attributes to implement are `batch_size`, `epoch`, `epoch_detail`, `is_new_epoch`, `iteration`, `__next__`, and `serialize`. Following is a code from the official example in the examples/ptb directory.

```
from __future__ import division

class ParallelSequentialIterator(chainer.dataset.Iterator):
    def __init__(self, dataset, batch_size, repeat=True):
        self.dataset = dataset
        self.batch_size = batch_size
        self.epoch = 0
        self.is_new_epoch = False
        self.repeat = repeat
        self.offsets = [i * len(dataset) // batch_size for i in range(batch_size)]
        self.iteration = 0

    def __next__(self):
        length = len(self.dataset)
        if not self.repeat and self.iteration * self.batch_size >= length:
            raise StopIteration
        cur_words = self.get_words()
        self.iteration += 1
```

(continues on next page)

(continued from previous page)

```
        next_words = self.get_words()

        epoch = self.iteration * self.batch_size // length
        self.is_new_epoch = self.epoch < epoch
        if self.is_new_epoch:
            self.epoch = epoch

        return list(zip(cur_words, next_words))

    @property
    def epoch_detail(self):
        return self.iteration * self.batch_size / len(self.dataset)

    def get_words(self):
        return [self.dataset[(offset + self.iteration) % len(self.dataset)]
                for offset in self.offsets]

    def serialize(self, serializer):
        self.iteration = serializer('iteration', self.iteration)
        self.epoch = serializer('epoch', self.epoch)

train_iter = ParallelSequentialIterator(train, 20)
val_iter = ParallelSequentialIterator(val, 1, repeat=False)
```

Although the code is slightly long, the idea is simple. First, this iterator creates `offsets` pointing to positions equally spaced within the whole sequence. The i-th examples of mini-batches refer the sequence with the i-th offset. The iterator returns a list of tuples of the current words and the next words. Each mini-batch is converted to a tuple of integer arrays by the `concat_examples` function in the standard updater (see the previous tutorial).

Backprop Through Time is implemented as follows.

```
class BPTTUpdater(training.updaters.StandardUpdater):

    def __init__(self, train_iter, optimizer, bprop_len):
        super(BPTTUpdater, self).__init__(train_iter, optimizer)
        self.bprop_len = bprop_len

    # The core part of the update routine can be customized by overriding.
    def update_core(self):
        loss = 0
        # When we pass one iterator and optimizer to StandardUpdater.__init__,
        # they are automatically named 'main'.
        train_iter = self.get_iterator('main')
        optimizer = self.get_optimizer('main')

        # Progress the dataset iterator for bprop_len words at each iteration.
        for i in range(self.bprop_len):
            # Get the next batch (a list of tuples of two word IDs)
            batch = train_iter.__next__()

            # Concatenate the word IDs to matrices and send them to the device
            # self.converter does this job
            # (it is chainer.dataset.concat_examples by default)
            x, t = self.converter(batch)

            # Compute the loss at this time step and accumulate it
            loss += optimizer.target(chainer.Variable(x), chainer.Variable(t))
```

(continues on next page)

(continued from previous page)

```
        optimizer.target.cleargrads()  # Clear the parameter gradients
        loss.backward()  # Backprop
        loss.unchain_backward()  # Truncate the graph
        optimizer.update()  # Update the parameters

updater = BPTTUpdater(train_iter, optimizer, bprop_len)  # instantiation
```

In this case, we update the parameters on every `bprop_len` consecutive words. The call of `unchain_backward` cuts the history of computation accumulated to the LSTM links. The rest of the code for setting up Trainer is almost same as one given in the previous tutorial.

In this section we have demonstrated how to write recurrent nets in Chainer and some fundamental techniques to manage the history of computation (a.k.a. computational graph). The example in the examples/ptb directory implements truncated backprop learning of a LSTM language model from the Penn Treebank corpus. In the next section, we will review how to use GPU(s) in Chainer.

3.6 RNN Language Models

3.6.1 0. Introduction

The **language model** is modeling the probability of generating natural language sentences or documents. You can use the language model to estimate how natural a sentence or a document is. Also, with the language model, you can generate new sentences or documents.

Let's start with modeling the probability of generating sentences. We represent a sentence as $\mathbf{X} = (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_T)$, in which $\mathbf{x}_t$ is a one-hot vector. Generally, $\mathbf{x}_0$ is the one-hot vector of **BOS** (beginning of sentence), and $\mathbf{x}_T$ is that of **EOS** (end of sentence).

A language model models the probability of a word occurrence under the condition of its previous words in a sentence. Let $\mathbf{X}_{[i,j]}$ be $(\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_j)$, the occurrence probability of sentence $\mathbf{X}$ can be represented as follows:

$$P(\mathbf{X}) = P(\mathbf{x}_0) \prod_{t=1}^{T} P(\mathbf{x}_t | \mathbf{X}_{[0,t-1]})$$

So, the language model $P(\mathbf{X})$ can be decomposed into word probabilities conditioned with its previous words. In this tutorial, we model $P(\mathbf{x}_t | \mathbf{X}_{[0,t-1]})$ with a recurrent neural network to obtain a language model $P(\mathbf{X})$.

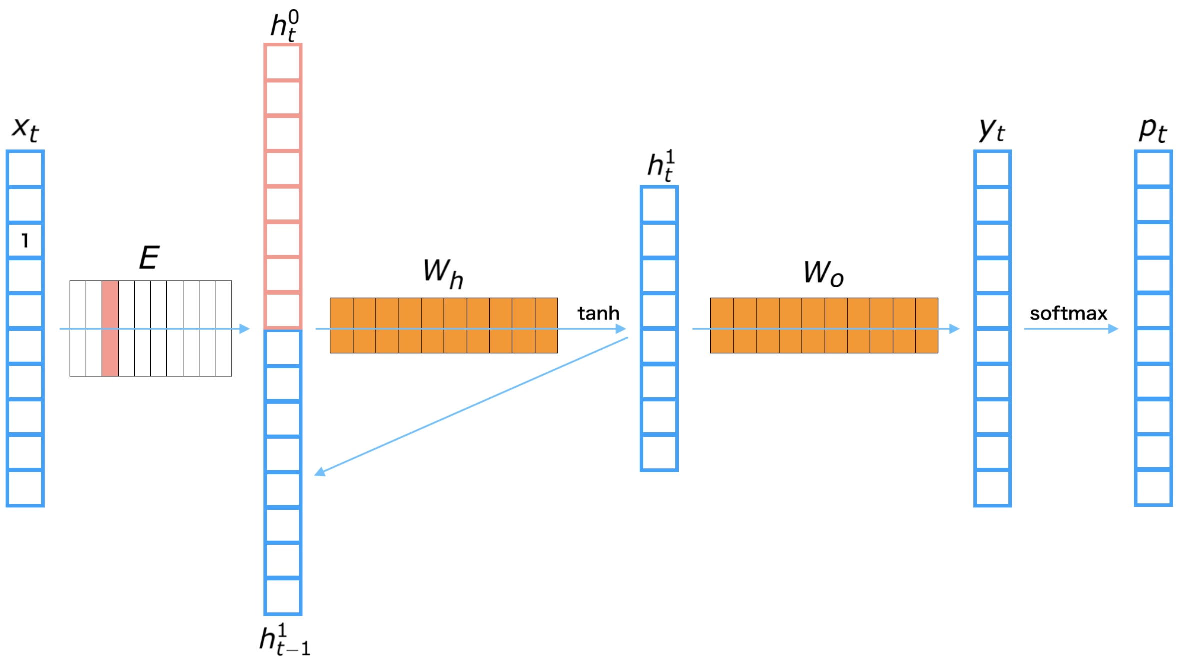
3.6.2 1. Basic Idea of Recurrent Neural Net Language Model

1.1 Recurrent Neural Net Language Model

Recurrent Neural Net Language Model (RNNLM) is a type of neural net language models which contains the RNNs in the network. Since an RNN can deal with the variable length inputs, it is suitable for modeling the sequential data such as sentences in natural language.

We show one layer of an RNNLM with these parameters.

Symbol	Definition
$\mathbf{x}_t$	the one-hot vector of t-th word
$\mathbf{y}_t$	the t-th output
$\mathbf{h}_t^{(i)}$	the t-th hidden layer of i-th layer
$\mathbf{p}_t$	the next word's probability of t-th word
$\mathbf{E}$	Embedding matrix
$\mathbf{W}_h$	Hidden layer matrix
$\mathbf{W}_o$	Output layer matrix



The process to get a next word prediction from i-th input word $\mathbf{x}_t$

1. Get the embedding vector: $\mathbf{h}_t^{(0)} = \mathbf{E}\mathbf{x}_t$
2. Calculate the hidden layer: $\mathbf{h}_t^{(1)} = \tanh\left(\mathbf{W}_h \left[\begin{array}{c}\mathbf{h}_t^{(0)} \\ \mathbf{h}_{t-1}^{(1)}\end{array}\right]\right)$
3. Calculate the output layer: $\mathbf{y}_t = \mathbf{W}_o\mathbf{h}_t^{(1)}$
4. Transform to probability: $\mathbf{p}_t = \mathrm{softmax}(\mathbf{y}_t)$

Note:

- Note that $\tanh$ in the above equation is applied to the input vector in element-wise manner.
- Note that $\left[\begin{array}{c}\mathbf{a} \\ \mathbf{b}\end{array}\right]$ denotes a concatenated vector of $\mathbf{a}$ and $\mathbf{b}$.
- Note that $\mathrm{softmax}$ in the above equation converts an arbitrary real vector to a probability vector which the summation over all elements is 1.

1.2 Perplexity (Evaluation of the language model)

Perplexity is the common evaluation metric for a language model. Generally, it measures how well the proposed probability model $P_{\rm model}(\mathbf{X})$ represents the target data $P^*(\mathbf{X})$. Let a validation dataset be $D = \{\mathbf{X}^{(n)}\}_{n=1}^{|D|}$, which is a set of sentences, where the n-th sentence length is $T^{(n)}$, and the vocabulary size of this dataset is $|\mathcal{V}|$, the perplexity is represented as follows:

$$b^z \quad s.t. \quad z = -\frac{1}{|\mathcal{V}|} \sum_{n=1}^{|D|} \sum_{t=1}^{T^{(n)}} \log_b P_{\rm model}(\mathbf{x}_t^{(n)}, \mathbf{X}_{[a,t-1]}^{(n)})$$

We usually use $b = 2$ or $b = e$. The perplexity shows how much varied the predicted distribution for the next word is. When a language model represents the dataset well, it should show a high probability only for the correct next word, so that the entropy should be high. In the above equation, the sign is reversed, so that smaller perplexity means better model.

During training, we minimize the below cross entropy:

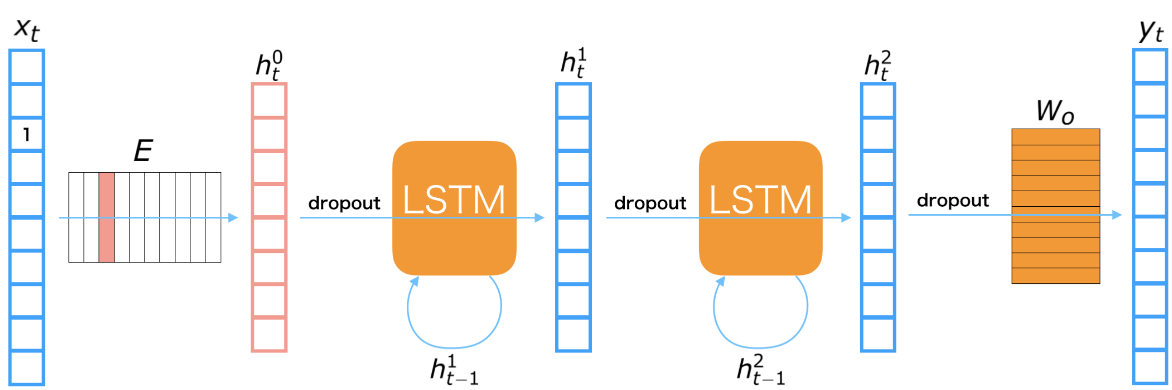
$$\mathcal{H}(\hat{P}, P_{\rm model}) = -\hat{P}(\mathbf{X}) \log P_{\rm model}(\mathbf{X})$$

where $\hat{P}$ is the empirical distribution of a sequence in the training dataset.

3.6.3 2. Implementation of Recurrent Neural Net Language Model

There is an example of RNN language model in the official repository, so we will explain how to implement a RNNLM in Chainer based on that: examples/ptb

2.1 Model Overview



The RNNLM used in this notebook is depicted in the above figure. The symbols appeared in the figure are defined as follows:

Symbol	Definition
$\mathbf{x}_t$	the one-hot vector of t-th word
$\mathbf{y}_t$	the t-th output
$\mathbf{h}_t^{(i)}$	the t-th hidden layer of i-th layer
$\mathbf{p}_t$	the next word's probability of t-th word
$\mathbf{E}$	Embedding matrix
$\mathbf{W}_h$	Hidden layer matrix
$\mathbf{W}_o$	Output layer matrix

LSTMs (long short-term memory) are used for the connection of hidden layers. A LSTM is one of major recurrent neural net modules. It is designed for remembering the long-term memory, so that it should be able to consider relationships of distant words, such that a word at beginning of sentence and it at the end. We also use **Dropout** before both LSTMs and linear transformations. Dropout is one of regularization techniques for preventing overfitting on training dataset.

2.2 Step-by-step Implementation

2.2.1 Import Package

First, let's import necessary packages.

Listing 11: train_ptb.py

```
"""
from __future__ import division
import argparse
import sys

import numpy as np
```

2.2.2 Define Training Settings

Define all training settings here.

Listing 12: train_ptb.py

```
parser.add_argument('--batchsize', '-b', type=int, default=20,
                    help='Number of examples in each mini-batch')
parser.add_argument('--bproplen', '-l', type=int, default=35,
                    help='Number of words in each mini-batch '
                         '(= length of truncated BPTT)')
parser.add_argument('--epoch', '-e', type=int, default=39,
                    help='Number of sweeps over the dataset to train')
parser.add_argument('--device', '-d', type=str, default='-1',
                    help='Device specifier. Either ChainerX device '
                    'specifier or an integer. If non-negative integer, '
                    'CuPy arrays with specified device id are used. If '
                    'negative integer, NumPy arrays are used')
parser.add_argument('--gradclip', '-c', type=float, default=5,
                    help='Gradient norm threshold to clip')
parser.add_argument('--out', '-o', default='result',
```

(continues on next page)

(continued from previous page)

```
                    help='Directory to output the result')
parser.add_argument('--resume', '-r', type=str,
                    help='Resume the training from snapshot')
parser.add_argument('--test', action='store_true',
                    help='Use tiny datasets for quick tests')
parser.set_defaults(test=False)
parser.add_argument('--unit', '-u', type=int, default=650,
                    help='Number of LSTM units in each layer')
parser.add_argument('--model', '-m', default='model.npz',
                    help='Model file name to serialize')
```

2.2.3 Define Network Structure

An RNNLM written in Chainer is shown below. It implements the model depicted in the above figure.

Listing 13: train_ptb.py

```
class RNNForLM(chainer.Chain):

    def __init__(self, n_vocab, n_units):
        super(RNNForLM, self).__init__()
        with self.init_scope():
            self.embed = L.EmbedID(n_vocab, n_units)
            self.l1 = L.LSTM(n_units, n_units)
            self.l2 = L.LSTM(n_units, n_units)
            self.l3 = L.Linear(n_units, n_vocab)

        for param in self.params():
            param.array[...] = np.random.uniform(-0.1, 0.1, param.shape)

    def reset_state(self):
        self.l1.reset_state()
        self.l2.reset_state()

    def forward(self, x):
        h0 = self.embed(x)
        h1 = self.l1(F.dropout(h0))
        h2 = self.l2(F.dropout(h1))
        y = self.l3(F.dropout(h2))
        return y
```

- When we instantiate this class for making a model, we give the vocabulary size to `n_vocab` and the size of hidden vectors to `n_units`.
- This network uses *chainer.links.LSTM*, *chainer.links.Linear*, and *chainer.functions.dropout* as its building blocks. All the layers are registered and initialized in the context with `self.init_scope()`.
- You can access all the parameters in those layers by calling `self.params()`.
- In the constructor, it initializes all parameters with values sampled from a uniform distribution $U(-1, 1)$.
- The `forward` method takes an word ID `x`, and calculates the word probability vector for the next word by forwarding it through the network, and returns the output.
- Note that the word ID `x` is automatically converted to a $|\mathcal{V}|$-dimensional one-hot vector and then multiplied with the input embedding matrix in `self.embed(x)` to obtain an embed vector `h0` at the first line of `forward`.

2.2.4 Load the Penn Tree Bank Long Word Sequence Dataset

In this notebook, we use Penn Tree Bank dataset that contains number of sentences. Chainer provides an utility function to obtain this dataset from server and convert it to a long single sequence of word IDs. `chainer.datasets.get_ptb_words()` actually returns three separated datasets which are for train, validation, and test.

Let's download and make dataset objects using it:

Listing 14: train_ptb.py

```
# Load the Penn Tree Bank long word sequence dataset
train, val, test = chainer.datasets.get_ptb_words()
```

2.2.5 Define Iterator for Making a Mini-batch from the Dataset

Dataset iterator creates a mini-batch of couple of words at different positions, namely, pairs of current word and its next word. Each example is a part of sentences starting from different offsets equally spaced within the whole sequence.

Listing 15: train_ptb.py

```
class ParallelSequentialIterator(chainer.dataset.Iterator):

    def __init__(self, dataset, batch_size, repeat=True):
        super(ParallelSequentialIterator, self).__init__()
        self.dataset = dataset
        self.batch_size = batch_size  # batch size
        self.repeat = repeat
        length = len(dataset)
        # Offsets maintain the position of each sequence in the mini-batch.
        self.offsets = [i * length // batch_size for i in range(batch_size)]
        self.reset()

    def reset(self):
        # Number of completed sweeps over the dataset. In this case, it is
        # incremented if every word is visited at least once after the last
        # increment.
        self.epoch = 0
        # True if the epoch is incremented at the last iteration.
        self.is_new_epoch = False
        # NOTE: this is not a count of parameter updates. It is just a count of
        # calls of ``__next__``.
        self.iteration = 0
        # use -1 instead of None internally
        self._previous_epoch_detail = -1.

    def __next__(self):
        # This iterator returns a list representing a mini-batch. Each item
        # indicates a different position in the original sequence. Each item is
        # represented by a pair of two word IDs. The first word is at the
        # "current" position, while the second word at the next position.
        # At each iteration, the iteration count is incremented, which pushes
        # forward the "current" position.
        length = len(self.dataset)
        if not self.repeat and self.iteration * self.batch_size >= length:
            # If not self.repeat, this iterator stops at the end of the first
```

(continues on next page)

(continued from previous page)

```
                # epoch (i.e., when all words are visited once).
                raise StopIteration
            cur_words = self.get_words()
            self._previous_epoch_detail = self.epoch_detail
            self.iteration += 1
            next_words = self.get_words()

            epoch = self.iteration * self.batch_size // length
            self.is_new_epoch = self.epoch < epoch
            if self.is_new_epoch:
                self.epoch = epoch

            return list(zip(cur_words, next_words))

        @property
        def epoch_detail(self):
            # Floating point version of epoch.
            return self.iteration * self.batch_size / len(self.dataset)

        @property
        def previous_epoch_detail(self):
            if self._previous_epoch_detail < 0:
                return None
            return self._previous_epoch_detail

        def get_words(self):
            # It returns a list of current words.
            return [self.dataset[(offset + self.iteration) % len(self.dataset)]
                    for offset in self.offsets]

        def serialize(self, serializer):
            # It is important to serialize the state to be recovered on resume.
            self.iteration = serializer('iteration', self.iteration)
            self.epoch = serializer('epoch', self.epoch)
            try:
                self._previous_epoch_detail = serializer(
                    'previous_epoch_detail', self._previous_epoch_detail)
            except KeyError:
                # guess previous_epoch_detail for older version
                self._previous_epoch_detail = self.epoch + \
                    (self.current_position - self.batch_size) / len(self.dataset)
                if self.epoch_detail > 0:
                    self._previous_epoch_detail = max(
                        self._previous_epoch_detail, 0.)
                else:
                    self._previous_epoch_detail = -1.
```

2.2.6 Define Updater

We use Backpropagation through time (BPTT) for optimize the RNNLM. BPTT can be implemented by overriding `update_core()` method of *StandardUpdater*. First, in the constructor of the `BPTTUpdater`, it takes `bprop_len` as an argument in addition to other arguments *StandardUpdater* needs. `bprop_len` defines the length of sequence T to calculate the loss:

$$\mathcal{L} = -\sum_{t=0}^{T}\sum_{n=1}^{|\mathcal{V}|}\hat{P}(\mathbf{x}_{t+1}^{(n)})\log P_{\mathrm{model}}(\mathbf{x}_{t+1}^{(n)} \mid \mathbf{x}_{t}^{(n)})$$

where $\hat{P}(\mathbf{x}_t^n)$ is a probability for n-th word in the vocabulary at the position t in the training data sequence.

Listing 16: train_ptb.py

```
class BPTTUpdater(training.updaters.StandardUpdater):

    def __init__(self, train_iter, optimizer, bprop_len, device):
        super(BPTTUpdater, self).__init__(
            train_iter, optimizer, device=device)
        self.bprop_len = bprop_len

    # The core part of the update routine can be customized by overriding.
    def update_core(self):
        loss = 0
        # When we pass one iterator and optimizer to StandardUpdater.__init__,
        # they are automatically named 'main'.
        train_iter = self.get_iterator('main')
        optimizer = self.get_optimizer('main')

        # Progress the dataset iterator for bprop_len words at each iteration.
        for i in range(self.bprop_len):
            # Get the next batch (a list of tuples of two word IDs)
            batch = train_iter.__next__()

            # Concatenate the word IDs to matrices and send them to the device
            # self.converter does this job
            # (it is chainer.dataset.concat_examples by default)
            x, t = self.converter(batch, self.device)

            # Compute the loss at this time step and accumulate it
            loss += optimizer.target(x, t)

        optimizer.target.cleargrads()  # Clear the parameter gradients
        loss.backward()  # Backprop
        loss.unchain_backward()  # Truncate the graph
        optimizer.update()  # Update the parameters
```

2.2.7 Define Evaluation Function (Perplexity)

Define a function to calculate the perplexity from the loss value. If we take e as b in the above definition of perplexity, calculating the perplexity is just to give the loss value to the power of e:

Listing 17: train_ptb.py

```
def compute_perplexity(result):
    result['perplexity'] = np.exp(result['main/loss'])
    if 'validation/main/loss' in result:
        result['val_perplexity'] = np.exp(result['validation/main/loss'])
```

2.2.8 Create Iterator

Here, the code below just creates iterator objects from dataset splits (train/val/test).

Listing 18: train_ptb.py

```
train_iter = ParallelSequentialIterator(train, args.batchsize)
val_iter = ParallelSequentialIterator(val, 1, repeat=False)
test_iter = ParallelSequentialIterator(test, 1, repeat=False)
```

2.2.9 Create RNN and Classification Model

Instantiate RNNLM model and wrap it with *chainer.links.Classifier* because it calculates softmax cross entropy as the loss.

Listing 19: train_ptb.py

```
rnn = RNNForLM(n_vocab, args.unit)
model = L.Classifier(rnn)
model.compute_accuracy = False  # we only want the perplexity
```

Note that *Classifier* computes not only the loss but also accuracy based on a given input/label pair. To learn the RNN language model, we only need the loss (cross entropy) in the *Classifier* because we calculate the perplexity instead of classification accuracy to check the performance of the model. So, we turn off computing the accuracy by giving False to `model.compute_accuracy` attribute.

2.2.10 Setup Optimizer

Prepare an optimizer. Here, we use *GradientClipping* to prevent gradient explosion. It automatically clips the gradient to be used to update the parameters in the model with given constant `gradclip`.

Listing 20: train_ptb.py

```
optimizer = chainer.optimizers.SGD(lr=1.0)
optimizer.setup(model)
optimizer.add_hook(chainer.optimizer_hooks.GradientClipping(args.gradclip))
```

2.2.11 Setup and Run Trainer

Let's make a trainer object and start the training! Note that we add an `eval_hook` to the *Evaluator* extension to reset the internal states before starting evaluation process. It can prevent to use training data during evaluating the model.

Listing 21: train_ptb.py

```
updater = BPTTUpdater(train_iter, optimizer, args.bproplen, device)
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)

eval_model = model.copy()  # Model with shared params and distinct states
```

(continues on next page)

(continued from previous page)

```
eval_rnn = eval_model.predictor
trainer.extend(extensions.Evaluator(
    val_iter, eval_model, device=device,
    # Reset the RNN state at the beginning of each evaluation
    eval_hook=lambda _: eval_rnn.reset_state()))

interval = 10 if args.test else 500
trainer.extend(extensions.LogReport(postprocess=compute_perplexity,
                                    trigger=(interval, 'iteration')))
trainer.extend(extensions.PrintReport(
    ['epoch', 'iteration', 'perplexity', 'val_perplexity']
), trigger=(interval, 'iteration'))
trainer.extend(extensions.ProgressBar(
    update_interval=1 if args.test else 10))
trainer.extend(extensions.snapshot())
trainer.extend(extensions.snapshot_object(
    model, 'model_iter_{.updater.iteration}'))
if args.resume is not None:
    chainer.serializers.load_npz(args.resume, trainer)

trainer.run()
```

2.2.12 Evaluate the trained model on test dataset

Let's see the perplexity on the test split. *Trainer*'s extension can be used as just a normal function outside of *Trainer*.

Listing 22: train_ptb.py

```
print('test')
eval_rnn.reset_state()
evaluator = extensions.Evaluator(test_iter, eval_model, device=device)
result = evaluator()
print('test perplexity: {}'.format(np.exp(float(result['main/loss']))))
```

2.3 Run Example

2.3.1 Training the model

You can train the model with the script: examples/ptb/train_ptb.py

```
$ pwd
/root2chainer/chainer/examples/ptb
$ python train_ptb.py --test  # run by test mode. If you want to use all data, remove
↪"--test".
Downloading from https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb.
↪train.txt...
Downloading from https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb.
↪valid.txt...
Downloading from https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb.
↪test.txt...
```

(continues on next page)

(continued from previous page)

```
#vocab = 10000
test
test perplexity: 29889.9857364
```

2.3.2 Generating sentences

You can generate the sentence which starts with a word in the vocabulary. In this example, we generate a sentence which starts with the word apple. We use the script in the PTB example of the official repository: examples/ptb/gentxt.py

```
$ pwd
/root2chainer/chainer/examples/ptb
$ python gentxt.py -m model.npz -p apple
apple a new u.s. economist with <unk> <unk> fixed more than to N the company said who␣
↪is looking back to
```

3.7 Word2Vec: Obtain word embeddings

3.7.1 0. Introduction

Word2vec is the tool for generating the distributed representation of words, which is proposed by Mikolov et al[1]. When the tool assigns a real-valued vector to each word, the closer the meanings of the words, the greater similarity the vectors will indicate.

Distributed representation means assigning a real-valued vector for each word and representing the word by the vector. When representing a word by distributed representation, we call the **word embeddings**. In this tutorial, we aim at explaining how to get the word embeddings from Penn Tree Bank dataset.

Let's think about what the meaning of word is. Since we are human, we can understand that the words "animal" and "dog" are deeply related each other. But what information will Word2vec use to learn the vectors for words? The words "animal" and "dog" should have similar vectors, but the words "food" and "dog" should be far from each other. How to know the features of those words automatically?

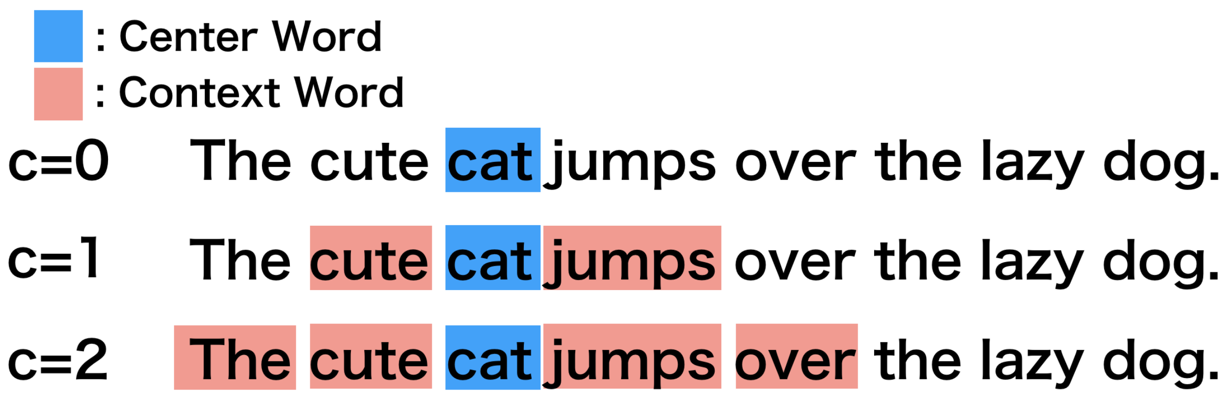
3.7.2 1. Basic Idea

Word2vec learns the similarity of word meanings from simple information. It learns the representation of words from sentences. The core idea is based on the assumption that the meaning of a word is affected by the words around it. This idea follows **distributional hypothesis**[2].

The word we focus on to learn its representation is called **center word**, and the words around it are called **context words**. The window size C determines the number of context words which is considered.

Here, let's see the algorithm by using an example sentence: "**The cute cat jumps over the lazy dog.**".

- All of the following figures consider "cat" as the center word.
- According to the window size C, you can see that the number of context words is changed.



3.7.3 2. Main Algorithm

Word2vec, the tool for creating the word embeddings, is actually built with two models, which are called **Skip-gram** and **CBoW**.

To explain the models with the figures below, we will use the following symbols.

Symbol	Definition
$\|\mathcal{V}\|$	The size of vocabulary
D	The size of embedding vector
$\mathbf{v}_t$	A one-hot center word vector
$V_{t\pm C}$	A set of $2C$ context vectors around $\mathbf{v}_t$, namely, $\{\mathbf{v}_{t+c}\}_{c=-C}^{C} \backslash \mathbf{v}_t$
$\mathbf{l}_H$	An embedding vector of an input word vector
$\mathbf{l}_O$	An output vector of the network
$\mathbf{W}_H$	The embedding matrix for inputs
$\mathbf{W}_O$	The embedding matrix for outputs

Note: Using **negative sampling** or **hierarchical softmax** for the loss function is very common, however, in this tutorial, we will use the **softmax over all words** and skip the other variants for the sake of simplicity.

2.1 Skip-gram

This model learns to predict context words $V_{t\pm C}$ when a center word $\mathbf{v}_t$ is given. In the model, each row of the embedding matrix for input $\mathbf{W}_H$ becomes a word embedding of each word.

When you input a center word $\mathbf{v}_t$ into the network, you can predict one of context words $\hat{\mathbf{v}}_{t+c} \in V_{t\pm C}$ as follows:

1. Calculate an embedding vector of the input center word vector: $\mathbf{l}_H = \mathbf{W}_H \mathbf{v}_t$
2. Calculate an output vector of the embedding vector: $\mathbf{l}_O = \mathbf{W}_O \mathbf{l}_H$
3. Calculate a probability vector of a context word: $\hat{\mathbf{v}}_{t+c} = \text{softmax}(\mathbf{l}_O)$

Each element of the $|\mathcal{V}|$-dimensional vector $\hat{\mathbf{v}}_{t+c}$ is a probability that a word in the vocabulary turns out to be a context word at position c. So, the probability $p(\mathbf{v}_{t+c}|\mathbf{v}_t)$ can be estimated by a dot product of the one-hot vector $\mathbf{v}_{t+c}$ which represents the actual word at the position c and the output vector $\hat{\mathbf{v}}_{t+c}$.

$$p(\mathbf{v}_{t+c}|\mathbf{v}_t) = \mathbf{v}_{t+c}^T \hat{\mathbf{v}}_{t+c}$$

The loss function to predict all the context words $V_{t\pm C}$ given a center word $\mathbf{v}_t$ is defined as follows:

$$\begin{aligned} L(V_{t\pm C}|\mathbf{v}_t; \mathbf{W}_H, \mathbf{W}_O) &= \sum_{V_{t\pm C}} -\log\left(p(\mathbf{v}_{t+c} \mid \mathbf{v}_t)\right) \\ &= \sum_{V_{t\pm C}} -\log(\mathbf{v}_{t+c}^T \hat{\mathbf{v}}_{t+c}) \end{aligned}$$

2.2 Continuous Bag of Words (CBoW)

This model learns to predict center word $\mathbf{v}_t$ when context words $V_{t\pm C}$ is given. When you give a set of context words $V_{t\pm C}$ to the network, you can estimate the probability of the center word $\hat{\mathbf{v}}_t$ as follows:

1. Calculate a mean embedding vector over all context words: $\mathbf{l}_H = \frac{1}{2C} \sum_{V_{t\pm C}} \mathbf{W}_H \mathbf{v}_{t+c}$
2. Calculate an output vector of the embedding vector: $\mathbf{l}_O = \mathbf{W}_O \mathbf{l}_H$
3. Calculate a probability vector of a center word: $\hat{\mathbf{v}}_t = \text{softmax}(\mathbf{l}_O)$

Each element of the $|\mathcal{V}|$-dimensional vector $\hat{\mathbf{v}}_t$ is a probability that a word in the vocabulary turns out to be a center word. So, the probability $p(\mathbf{v}_t|V_{t\pm C})$ can be estimated by a dot product of the one-hot vector $\mathbf{v}_t$ which represents the actual center word and the output vector $\hat{\mathbf{v}}_t$.

$$p(\mathbf{v}_t|V_{t\pm C}) = \mathbf{v}_t^T \hat{\mathbf{v}}_t$$

The loss function to predict the center word $\mathbf{v}_t$ given context words $V_{t\pm C}$ is defined as follows:

$$\begin{aligned} L(\mathbf{v}_t|V_{t\pm C}; \mathbf{W}_H, \mathbf{W}_O) &= -\log\left(p(\mathbf{v}_t \mid V_{t\pm C})\right) \\ &= -\log(\mathbf{v}_t^T \hat{\mathbf{v}}_t) \end{aligned}$$

3.7.4 3. Details of Skip-gram

In this tutorial, we mainly explain Skip-gram model because

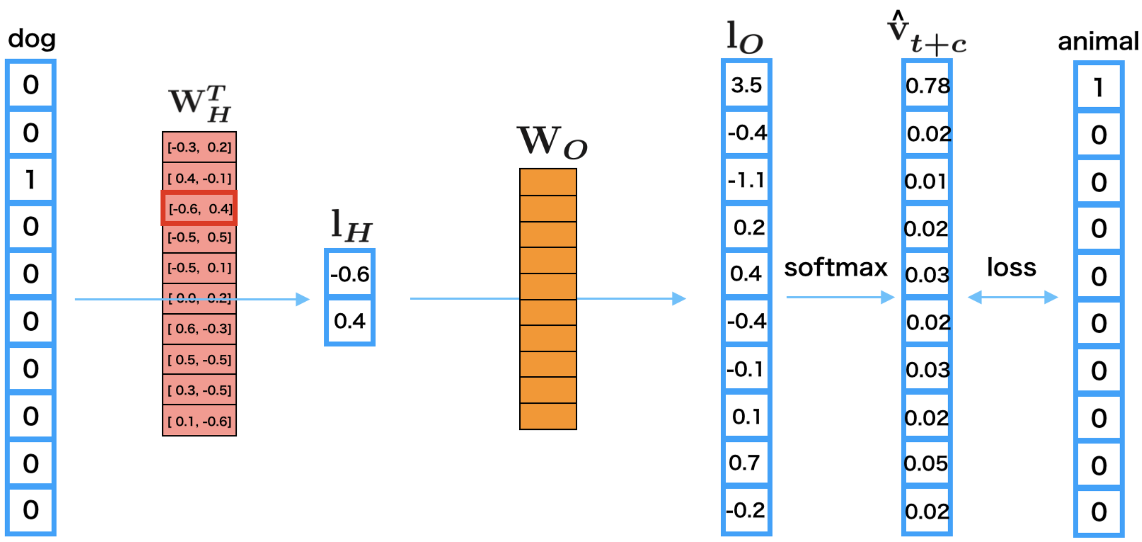
1. It is easier to understand the algorithm than CBoW.
2. Even if the number of words increases, the accuracy is largely maintained. So, it is more scalable.

So, let's think about a concrete example of calculating Skip-gram under this setup:

- The size of vocabulary $|\mathcal{V}|$ is 10.
- The size of embedding vector D is 2.
- Center word is "dog".
- Context word is "animal".

Since there should be more than one context word, repeat the following process for each context word.

1. The one-hot vector of "dog" is `[0 0 1 0 0 0 0 0 0 0]` and you input it as the center word.
2. The third row of embedding matrix $\mathbf{W}_H$ is used for the word embedding of "dog" $\mathbf{l}_H$.
3. Then, multiply $\mathbf{W}_O$ with $\mathbf{l}_H$ to obtain the output vector $\mathbf{l}_O$.
4. Give $\mathbf{l}_O$ to the softmax function to make it a predicted probability vector $\hat{\mathbf{v}}_{t+c}$ for a context word at the position c.
5. Calculate the error between $\hat{\mathbf{v}}_{t+c}$ and the one-hot vector of "animal"; `[1 0 0 0 0 0 0 0 0 0 0]`.
6. Propagate the error back to the network to update the parameters.



3.7.5 4. Implementation of Skip-gram in Chainer

There is an example of Word2vec in the official repository of Chainer, so we will explain how to implement Skip-gram based on this: examples/word2vec

4.1 Preparation

First, let's import necessary packages:

Listing 23: train_word2vec.py

```
import argparse
import collections
import os
import six
import warnings

import numpy as np

import chainer
from chainer.backends import cuda
import chainer.functions as F
import chainer.initializers as I
import chainer.links as L
import chainer.optimizers as O
from chainer import reporter
```

4.2 Define a Skip-gram model

Next, let's define a network for Skip-gram.

Listing 24: train_word2vec.py

```
class SkipGram(chainer.Chain):
    """Definition of Skip-gram Model"""

    def __init__(self, n_vocab, n_units, loss_func):
        super(SkipGram, self).__init__()

        with self.init_scope():
            self.embed = L.EmbedID(
                n_vocab, n_units, initialW=I.Uniform(1. / n_units))
            self.loss_func = loss_func

    def forward(self, x, contexts):
        e = self.embed(contexts)
        batch_size, n_context, n_units = e.shape
        x = F.broadcast_to(x[:, None], (batch_size, n_context))
        e = F.reshape(e, (batch_size * n_context, n_units))
        x = F.reshape(x, (batch_size * n_context,))
        loss = self.loss_func(e, x)
        reporter.report({'loss': loss}, self)
        return loss
```

Listing 25: train_word2vec.py

```
class SoftmaxCrossEntropyLoss(chainer.Chain):
    """Softmax cross entropy loss function preceded by linear transformation.

    """

    def __init__(self, n_in, n_out):
        super(SoftmaxCrossEntropyLoss, self).__init__()
        with self.init_scope():
            self.out = L.Linear(n_in, n_out, initialW=0)

    def forward(self, x, t):
        return F.softmax_cross_entropy(self.out(x), t)
```

Note:

- The weight matrix `self.embed.W` is the embedding matrix for input vector `x`.
- The function call `forward` takes the word ID of a center word `x` and word IDs of context words contexts as inputs, and outputs the error calculated by the loss function `loss_func` s.t. `SoftmaxCrossEntropyLoss`.
- Note that the initial shape of `x` and contexts are `(batch_size,)` and `(batch_size, n_context)`, respectively.
- The `batch_size` means the size of mini-batch, and `n_context` means the number of context words.

First, we obtain the embedding vectors of contexts by `e = self.embed(contexts)`. Then `F.broadcast_to(x[:, None], (batch_size, n_context))` performs broadcasting of `x` (its shape is `(batch_size,)`) to `(batch_size, n_context)` by copying the same value `n_context` time to fill the second axis, and then the broadcasted `x` is reshaped into 1-D vector `(batchsize * n_context,)` while `e` is reshaped to `(batch_size * n_context, n_units)`. In Skip-gram model, predicting a context word from the center word is the same as predicting the center word from a context word because the center word is always a context word when considering the context word as a center word. So, we create `batch_size * n_context` center word predictions by applying `self.out` linear layer to the embedding vectors of context words. Then, calculate softmax cross entropy between the broadcasted center word ID x and the predictions.

4.3 Prepare dataset and iterator

Let's retrieve the Penn Tree Bank (PTB) dataset by using Chainer's dataset utility `get_ptb_words()` method.

```
train, val, _ = chainer.datasets.get_ptb_words()
counts = collections.Counter(train)
```

Then define an iterator to make mini-batches that contain a set of center words with their context words. `train` and `val` means training data and validation data. Each data contains the list of Document IDs:

```
>>> train
array([ 0,   1,   2, ..., 39, 26, 24], dtype=int32)
>>> val
array([2211,  396, 1129, ...,  108,    27,    24], dtype=int32)
```

Listing 26: train_word2vec.py

```
class WindowIterator(chainer.dataset.Iterator):
    """Dataset iterator to create a batch of sequences at different positions.

    This iterator returns a pair of the current words and the context words.
    """

    def __init__(self, dataset, window, batch_size, repeat=True):
        self.dataset = np.array(dataset, np.int32)
        self.window = window  # size of context window
        self.batch_size = batch_size
        self._repeat = repeat
        # order is the array which is shuffled ``[window, window + 1, ...,
        # len(dataset) - window - 1]``
        self.order = np.random.permutation(
            len(dataset) - window * 2).astype(np.int32)
        self.order += window
        self.current_position = 0
        # Number of completed sweeps over the dataset. In this case, it is
        # incremented if every word is visited at least once after the last
        # increment.
        self.epoch = 0
        # True if the epoch is incremented at the last iteration.
        self.is_new_epoch = False

    def __next__(self):
        """This iterator returns a list representing a mini-batch.

        Each item indicates a different position in the original sequence.
        """
        if not self._repeat and self.epoch > 0:
            raise StopIteration

        i = self.current_position
        i_end = i + self.batch_size
        position = self.order[i:i_end]
```

(continues on next page)

(continued from previous page)

```
        w = np.random.randint(self.window - 1) + 1
        offset = np.concatenate([np.arange(-w, 0), np.arange(1, w + 1)])
        pos = position[:, None] + offset[None, :]
        contexts = self.dataset.take(pos)
        center = self.dataset.take(position)

        if i_end >= len(self.order):
            np.random.shuffle(self.order)
            self.epoch += 1
            self.is_new_epoch = True
            self.current_position = 0
        else:
            self.is_new_epoch = False
            self.current_position = i_end

        return center, contexts

    @property
    def epoch_detail(self):
        return self.epoch + float(self.current_position) / len(self.order)

    def serialize(self, serializer):
        self.current_position = serializer('current_position',
                                           self.current_position)
        self.epoch = serializer('epoch', self.epoch)
        self.is_new_epoch = serializer('is_new_epoch', self.is_new_epoch)
        if self.order is not None:
            serializer('order', self.order)
```

- In the constructor, we create an array `self.order` which denotes shuffled indices of `[window, window + 1, ..., len(dataset) - window - 1]` in order to choose a center word randomly from dataset in a mini-batch.
- The iterator definition `__next__` returns `batch_size` sets of center word and context words.
- The code `self.order[i:i_end]` returns the indices for a set of center words from the random-ordered array `self.order`. The center word IDs center at the random indices are retrieved by `self.dataset.take`.
- `np.concatenate([np.arange(-w, 0), np.arange(1, w + 1)])` creates a set of offsets to retrieve context words from the dataset.
- The code `position[:, None] + offset[None, :]` generates the indices of context words for each center word index in position. The context word IDs context are retrieved by `self.dataset.take`.

4.4 Prepare model, optimizer, and updater

Listing 27: train_word2vec.py

```
    model = SkipGram(n_vocab, args.unit, loss_func)
```

Listing 28: train_word2vec.py

```
optimizer = O.Adam()
optimizer.setup(model)
```

Listing 29: train_word2vec.py

```
train_iter = WindowIterator(train, args.window, args.batchsize)
val_iter = WindowIterator(val, args.window, args.batchsize, repeat=False)

# Set up an updater
updater = training.updaters.StandardUpdater(
    train_iter, optimizer, converter=convert, device=device)
```

Listing 30: train_word2vec.py

```
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)

trainer.extend(extensions.Evaluator(
    val_iter, model, converter=convert, device=device))
trainer.extend(extensions.LogReport())
trainer.extend(extensions.PrintReport(
    ['epoch', 'main/loss', 'validation/main/loss']))
trainer.extend(extensions.ProgressBar())

trainer.extend(
    extensions.snapshot(filename='snapshot_epoch_{.updater.epoch}'),
    trigger=(args.snapshot_interval, 'epoch'))

if args.resume is not None:
    chainer.serializers.load_npz(args.resume, trainer)
trainer.run()
```

4.5 Start training

```
$ pwd
/root2chainer/chainer/examples/word2vec
$ python train_word2vec.py --test  # run by test mode. If you want to use all data,␣
↪remove "--test".
GPU: -1
# unit: 100
Window: 5
Minibatch-size: 1000
# epoch: 20
Training model: skipgram
Output type: hsm

n_vocab: 10000
data length: 100
epoch       main/loss   validation/main/loss
1           4233.75     2495.33
2           1411.14     4990.66
3           4233.11     1247.66
4           2821.66     4990.65
5           4231.94     1247.66
6           5642.04     2495.3
7           5640.82     4990.64
8           5639.31     2495.28
```

(continues on next page)

(continued from previous page)

```
9           2817.89     4990.62
10          1408.03     3742.94
11          5633.11     1247.62
12          4221.71     2495.21
13          4219.3      4990.56
14          4216.57     2495.16
15          4213.52     2495.12
16          5616.03     1247.55
17          5611.34     3742.78
18          2800.31     3742.74
19          1397.79     2494.95
20          2794.1      3742.66
```

4.5 Search the similar words

```
$ pwd
/root2chainer/chainer/examples/word2vec
$ python search.py
>> apple
query: apple
compaq: 0.6169619560241699
chip: 0.49579331278800964
retailer: 0.4904134273529053
maker: 0.4684058427810669
computer: 0.4652436673641205
>> animal
query: animal
beauty: 0.5680124759674072
human: 0.5404794216156006
insulin: 0.5365156531333923
cell: 0.5186758041381836
photographs: 0.5077002048492432
```

3.7.6 5. Reference

- [1] Mikolov, Tomas; et al. "Efficient Estimation of Word Representations in Vector Space". arXiv:1301.3781
- [2] Distributional Hypothesis

3.8 Write a Sequence to Sequence (seq2seq) Model

3.8.1 0. Introduction

The **sequence to sequence (seq2seq) model**[1][2] is a learning model that converts an input sequence into an output sequence. In this context, the **sequence** is a list of symbols, corresponding to the words in a sentence. The seq2seq model has achieved great success in fields such as machine translation, dialogue systems, question answering, and text summarization. All of these tasks can be regarded as the task to learn a model that converts an input sequence into an output sequence.

3.8.2 1. Basic Idea of Seq2seq Model

1.1 Overview of Seq2seq Model

The Notations of Sequence

The seq2seq model converts an input sequence into an output sequence. Let the input sequence and the output sequence be $\mathbf{X}$ and $\mathbf{Y}$. The i-th element of the input sequence is represented as $\mathbf{x}_i$, and the j-th element of the output sequence is also represented as $\mathbf{y}_j$. Generally, each of the $\mathbf{x}_i$ and the $\mathbf{y}_j$ is the one-hot vector of the symbols. For example, in natural language processing(NLP), the one-hot vector represents the word and its size becomes the vocabulary size.

Let's think about the seq2seq model in the context of NLP. Let the vocabulary of the inputs and the outputs be $\mathcal{V}^{(s)}$ and $\mathcal{V}^{(t)}$, all the elements $\mathbf{x}_i$ and $\mathbf{y}_j$ satisfy $\mathbf{x}_i \in \mathbb{R}^{|\mathcal{V}^{(s)}|}$ and $\mathbf{y}_i \in \mathbb{R}^{|\mathcal{V}^{(t)}|}$. The input sequence $\mathbf{X}$ and the output sequence $\mathbf{Y}$ are represented as the following equations:

$$\mathbf{X} = (\mathbf{x}_1, ..., \mathbf{x}_I) = (\mathbf{x}_i)_{i=1}^{I}$$
$$\mathbf{Y} = (\mathbf{y}_1, ..., \mathbf{y}_J) = (\mathbf{y}_j)_{j=1}^{J}$$

I and J are the length of the input sequence and the output sequence. Using the typical NLP notation, $\mathbf{y}_0$ is the one-hot vector of *BOS*, which is the virtual word representing the beginning of the sentence, and $\mathbf{y}_{J+1}$ is that of *EOS*, which is the virtual word representing the end of the sentence.

The Notations of Conditional Probability $P(\mathbf{Y}|\mathbf{X})$

Next, let's think about the conditional probability $P(\mathbf{Y}|\mathbf{X})$ generating the output sequence $\mathbf{Y}$ when the input sequence $\mathbf{X}$ is given. The purpose of seq2seq model is modeling the probability $P(\mathbf{Y}|\mathbf{X})$. However, the seq2seq model does not model the probability $P(\mathbf{Y}|\mathbf{X})$ directly. Actually, it models the probability $P(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{X})$, which is the probability of generating the j-th element of the output sequence $\mathbf{y}_j$ given the $\mathbf{Y}_{<j}$ and $\mathbf{X}$. $\mathbf{Y}_{<j}$ means the output sequence from 1 to $j-1$, or $(\mathbf{y}_j)_{j=1}^{j-1}$. In this notation, you can write the model $P_\theta(\mathbf{Y}|\mathbf{X})$ with the product of $P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{X})$:

$$P_\theta(\mathbf{Y}|\mathbf{X}) = \prod_{j=1}^{J+1} P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{X})$$

Processing Steps in Seq2seq Model

Now, let's think about the processing steps in seq2seq model. The feature of seq2seq model is that it consists of the two processes:

1. The process that generates the fixed size vector $\mathbf{z}$ from the input sequence $\mathbf{X}$
2. The process that generates the output sequence $\mathbf{Y}$ from $\mathbf{z}$

In other words, the information of $\mathbf{X}$ is conveyed by $\mathbf{z}$, and $P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{X})$ is actually calculated by $P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{z})$.

First, we represent the process which generating $\mathbf{z}$ from $\mathbf{X}$ by the function Λ:

$$\mathbf{z} = \Lambda(\mathbf{X})$$

The function Λ may be the recurrent neural net such as LSTMs.

Second, we represent the process which generating $\mathbf{Y}$ from $\mathbf{z}$ by the following formula:

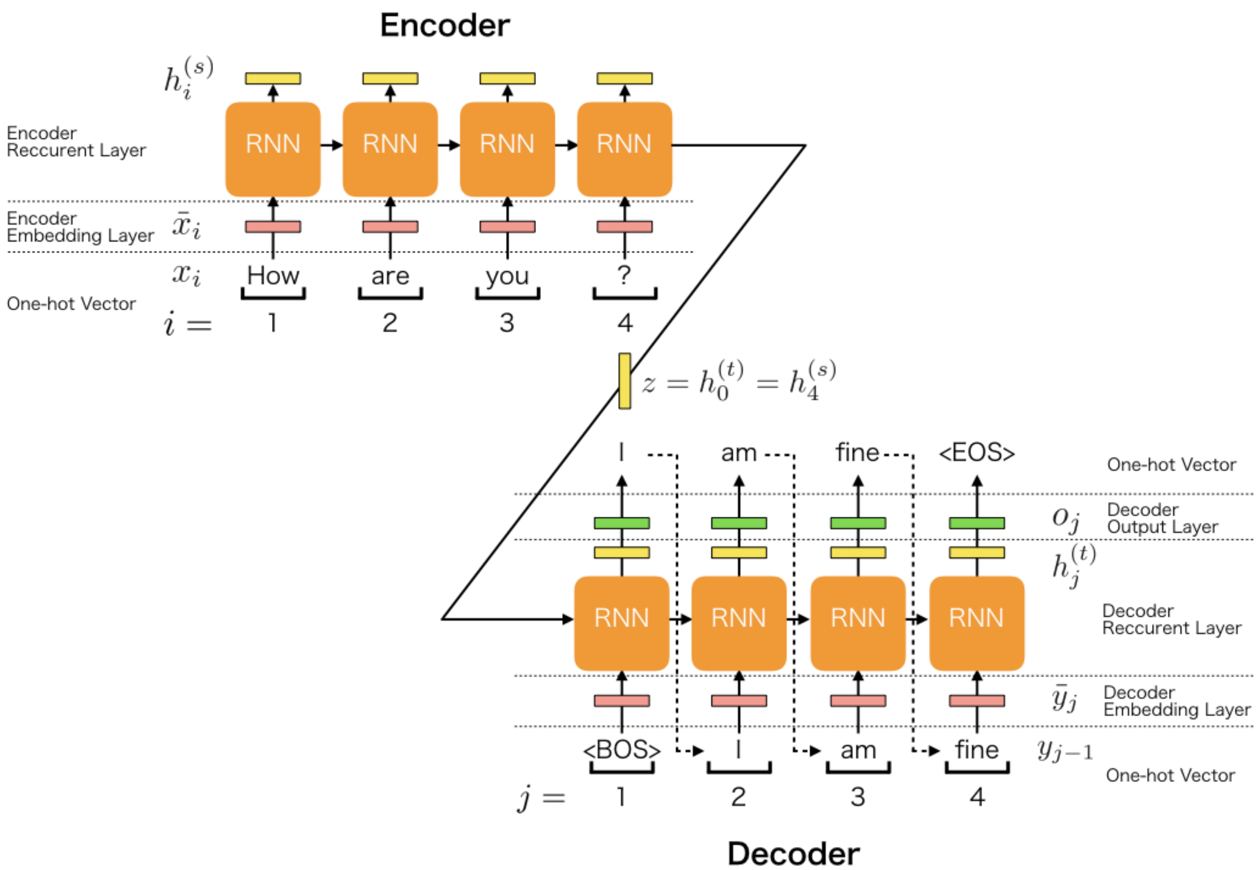
$$P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}, \mathbf{X}) = \Upsilon(\mathbf{h}_j^{(t)}, \mathbf{y}_j)$$
$$\mathbf{h}_j^{(t)} = \Psi(\mathbf{h}_{j-1}^{(t)}, \mathbf{y}_{j-1})$$

Ψ is the function to generate the hidden vectors $\mathbf{h}_j^{(t)}$, and Υ is the function to calculate the generative probability of the one-hot vector $\mathbf{y}_j$. When $j = 1$, $\mathbf{h}_{j-1}^{(t)}$ or $\mathbf{h}_0^{(t)}$ is $\mathbf{z}$ generated by $\Lambda(\mathbf{X})$, and $\mathbf{y}_{j-1}$ or $\mathbf{y}_0$ is the one-hot vector of *BOS*.

1.2 Model Architecture of Seq2seq Model

In this section, we describe the architecture of seq2seq model. To simplify the explanation, we use the most basic architecture. The architecture of seq2seq model can be separated to the five major roles.

1. Encoder Embedding Layer
2. Encoder Recurrent Layer
3. Decoder Embedding Layer
4. Decoder Recurrent Layer
5. Decoder Output Layer



The encoder consists of two layers: the embedding layer and the recurrent layer, and the decoder consists of three layers: the embedding layer, the recurrent layer, and the output layer.

In the explanation, we use the following symbols:

Symbol	Definition
H	the size of the hidden vector
D	the size of the embedding vector
$\mathbf{x}_i$	the one-hot vector of i-th word in the input sentence
$\bar{\mathbf{x}}_i$	the embedding vector of i-th word in the input sentence
$\mathbf{E}^{(s)}$	Embedding matrix of the encoder
$\mathbf{h}_i^{(s)}$	the i-th hidden vector of the encoder
$\mathbf{y}_j$	the one-hot vector of j-th word in the output sentence
$\bar{\mathbf{y}}_j$	the embedding vector of j-th word in the output sentence
$\mathbf{E}^{(t)}$	Embedding matrix of the decoder
$\mathbf{h}_j^{(t)}$	the j-th hidden vector of the decoder

1.2.1 Encoder Embedding Layer

The first layer, or the encoder embedding layer converts the each word in the input sentence to the embedding vector. When processing the i-th word in the input sentence, the input and the output of the layer are the following:

- The input is $\mathbf{x}_i$: the one-hot vector which represents i-th word
- The output is $\bar{\mathbf{x}}_i$: the embedding vector which represents i-th word

Each embedding vector is calculated by the following equation:

$$\bar{\mathbf{x}}_i = \mathbf{E}^{(s)}\mathbf{x}_i$$

$\mathbf{E}^{(s)} \in \mathbb{R}^{D \times |\mathcal{V}^{(s)}|}$ is the embedding matrix of the encoder.

1.2.2 Encoder Recurrent Layer

The encoder recurrent layer generates the hidden vectors from the embedding vectors. When processing the i-th embedding vector, the input and the output of the layer are the following:

- The input is $\bar{\mathbf{x}}_i$: the embedding vector which represents the i-th word
- The output is $\mathbf{h}_i^{(s)}$: the hidden vector of the i-th position

For example, when using the uni-directional RNN of one layer, the process can be represented as the following function $\Psi^{(s)}$:

$$\begin{aligned}\mathbf{h}_i^{(s)} &= \Psi^{(s)}(\bar{\mathbf{x}}_i, \mathbf{h}_{i-1}^{(s)}) \\ &= \tanh\left(\mathbf{W}^{(s)}\begin{bmatrix}\mathbf{h}_{i-1}^{(s)} \\ \bar{\mathbf{x}}_i\end{bmatrix} + \mathbf{b}^{(s)}\right)\end{aligned}$$

In this case, we use the $\tanh$ as the activation function.

1.2.3 Decoder Embedding Layer

The decoder embedding layer converts the each word in the output sentence to the embedding vector. When processing the j-th word in the output sentence, the input and the output of the layer are the following:

- The input is $\mathbf{y}_{j-1}$: the one-hot vector which represents the $(j-1)$-th word generated by the decoder output layer

- The output is $\bar{\mathbf{y}}_j$: the embedding vector which represents the $(j-1)$-th word

Each embedding vector is calculated by the following equation:

$$\bar{\mathbf{y}}_j = \mathbf{E}^{(t)}\mathbf{y}_{j-1}$$

$\mathbf{E}^{(t)} \in \mathbb{R}^{D\times|\mathcal{V}^{(t)}|}$ is the embedding matrix of the encoder.

1.2.4 Decoder Recurrent Layer

The decoder recurrent layer generates the hidden vectors from the embedding vectors. When processing the j-th embedding vector, the input and the output of the layer are the following:

- The input is $\bar{\mathbf{y}}_j$: the embedding vector
- The output is $\mathbf{h}_j^{(t)}$: the hidden vector of j-th position

For example, when using the uni-directional RNN of one layer, the process can be represented as the following function $\Psi^{(t)}$:

$$\begin{aligned}\mathbf{h}_j^{(t)} &= \Psi^{(t)}(\bar{\mathbf{y}}_j, \mathbf{h}_{j-1}^{(t)}) \\ &= \tanh\left(\mathbf{W}^{(t)}\begin{bmatrix}\mathbf{h}_{j-1}^{(t)} \\ \bar{\mathbf{y}}_j\end{bmatrix} + \mathbf{b}^{(t)}\right)\end{aligned}$$

In this case, we use the $\tanh$ as the activation function. And we must use the encoder's hidden vector of the last position as the decoder's hidden vector of first position as following:

$$\mathbf{h}_0^{(t)} = \mathbf{z} = \mathbf{h}_I^{(s)}$$

1.2.5 Decoder Output Layer

The decoder output layer generates the probability of the j-th word of the output sentence from the hidden vector. When processing the j-th embedding vector, the input and the output of the layer are the following:

- The input is $\mathbf{h}_j^{(t)}$: the hidden vector of j-th position
- The output is p_j : the probability of generating the one-hot vector $\mathbf{y}_j$ of the j-th word

$$\begin{aligned}p_j &= P_\theta(\mathbf{y}_j|\mathbf{Y}_{<j}) = \mathrm{softmax}(\mathbf{o}_j)\cdot\mathbf{y}_j \\ &= \mathrm{softmax}(\mathbf{W}^{(o)}\mathbf{h}_j^{(t)} + \mathbf{b}^{(o)})\cdot\mathbf{y}_j\end{aligned}$$

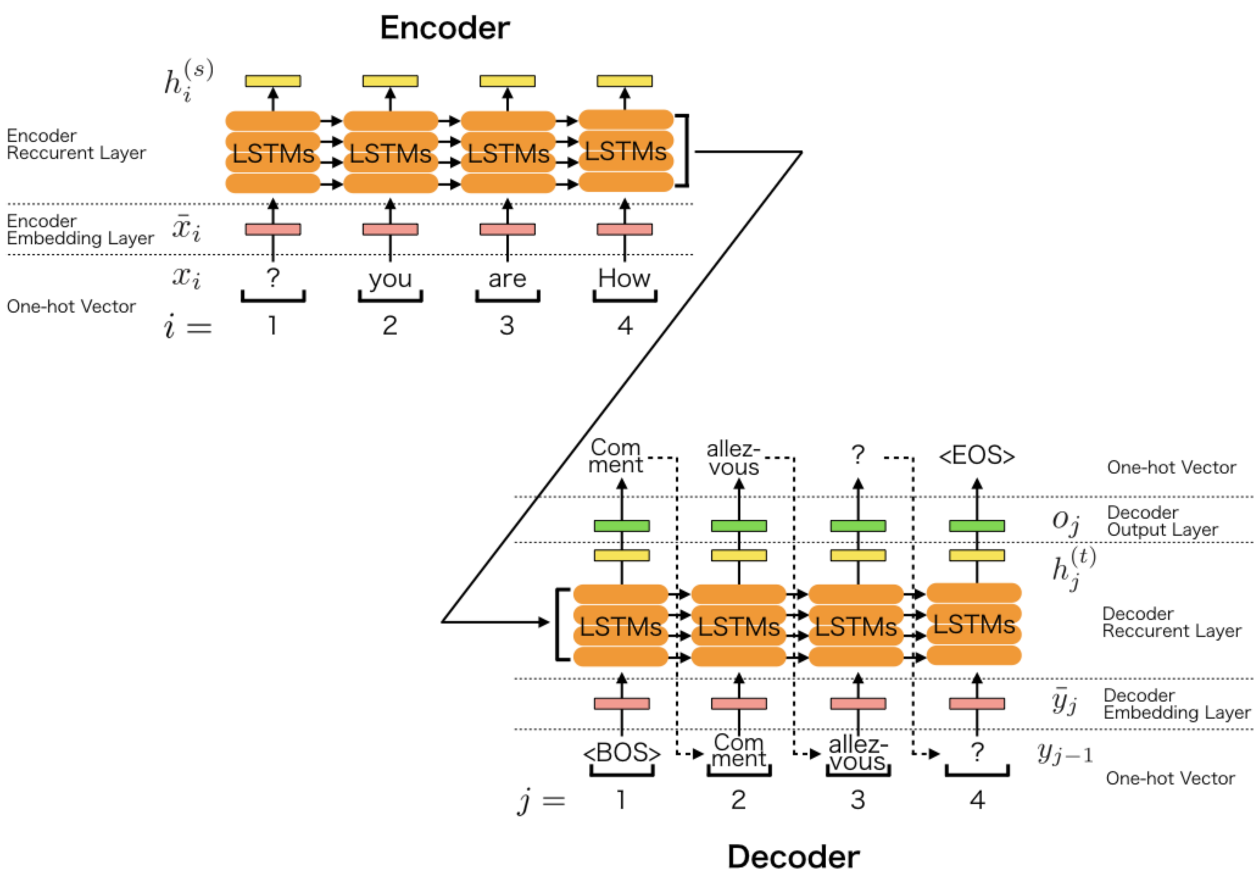
Note: There are a lot of varieties of seq2seq models. We can use the different RNN models in terms of: (1) directionality (unidirectional or bidirectional), (2) depth (single-layer or multi-layer), (3) type (a vanilla RNN, a Long Short-term Memory (LSTM), or a gated recurrent unit (GRU)), and (4) additional functionality (s.t. Attention Mechanism).

3.8.3 2. Implementation of Seq2seq Model

The official Chainer repository includes a neural machine translation example using the seq2seq model. We will now provide an overview of the example and explain its implementation in detail. chainer/examples/seq2seq

2.1 Model Overview

In this simple example, an input sequence is processed by a stacked **LSTM-RNN** (long short-term memory recurrent neural networks) and it is encoded as a fixed-size vector. The output sequence is also processed by another stacked LSTM-RNN. At decoding time, an output sequence is generated using argmax.



2.2 Step-by-step Implementation

2.2.1 Import Package

First, let's import necessary packages.

Listing 31: seq2seq.py

```
import io

from nltk.translate import bleu_score
import numpy
import progressbar
import six

import chainer
import chainer.functions as F
```

(continues on next page)

(continued from previous page)

```
import chainer.links as L
from chainer import training
```

2.2.2 Define Training Settings

Define all training settings here.

Listing 32: seq2seq.py

```
parser.add_argument('SOURCE', help='source sentence list')
parser.add_argument('TARGET', help='target sentence list')
parser.add_argument('SOURCE_VOCAB', help='source vocabulary file')
parser.add_argument('TARGET_VOCAB', help='target vocabulary file')
parser.add_argument('--validation-source',
                    help='source sentence list for validation')
parser.add_argument('--validation-target',
                    help='target sentence list for validation')
parser.add_argument('--batchsize', '-b', type=int, default=64,
                    help='number of sentence pairs in each mini-batch')
parser.add_argument('--epoch', '-e', type=int, default=20,
                    help='number of sweeps over the dataset to train')
parser.add_argument('--resume', '-r', type=str,
                    help='resume the training from snapshot')
parser.add_argument('--save', '-s', type=str,
                    help='save a snapshot of the training')
parser.add_argument('--unit', '-u', type=int, default=1024,
                    help='number of units')
parser.add_argument('--layer', '-l', type=int, default=3,
                    help='number of layers')
parser.add_argument('--use-dataset-api', default=False,
                    action='store_true',
                    help='use TextDataset API to reduce CPU memory usage')
parser.add_argument('--min-source-sentence', type=int, default=1,
                    help='minimium length of source sentence')
parser.add_argument('--max-source-sentence', type=int, default=50,
                    help='maximum length of source sentence')
parser.add_argument('--min-target-sentence', type=int, default=1,
                    help='minimium length of target sentence')
parser.add_argument('--max-target-sentence', type=int, default=50,
                    help='maximum length of target sentence')
parser.add_argument('--log-interval', type=int, default=200,
                    help='number of iteration to show log')
parser.add_argument('--validation-interval', type=int, default=4000,
                    help='number of iteration to evlauate the model '
                    'with validation dataset')
parser.add_argument('--device', '-d', type=str, default='-1',
                    help='Device specifier. Either ChainerX device '
                    'specifier or an integer. If non-negative integer, '
                    'CuPy arrays with specified device id are used. If '
                    'negative integer, NumPy arrays are used')
parser.add_argument('--out', '-o', default='result',
                    help='directory to output the result')
group = parser.add_argument_group('deprecated arguments')
group.add_argument('--gpu', '-g', dest='device',
                   type=int, nargs='?', const=0,
```

(continues on next page)

(continued from previous page)

```
                    help='GPU ID (negative value indicates CPU)')
```

2.2.3 Define Network Structure

The Chainer implementation of seq2seq is shown below. It implements the model depicted in the above figure.

Listing 33: seq2seq.py

```
class Seq2seq(chainer.Chain):

    def __init__(self, n_layers, n_source_vocab, n_target_vocab, n_units):
        super(Seq2seq, self).__init__()
        with self.init_scope():
            self.embed_x = L.EmbedID(n_source_vocab, n_units)
            self.embed_y = L.EmbedID(n_target_vocab, n_units)
            self.encoder = L.NStepLSTM(n_layers, n_units, n_units, 0.1)
            self.decoder = L.NStepLSTM(n_layers, n_units, n_units, 0.1)
            self.W = L.Linear(n_units, n_target_vocab)

        self.n_layers = n_layers
        self.n_units = n_units

    def forward(self, xs, ys):
        xs = [x[::-1] for x in xs]

        eos = self.xp.array([EOS], numpy.int32)
        ys_in = [F.concat([eos, y], axis=0) for y in ys]
        ys_out = [F.concat([y, eos], axis=0) for y in ys]

        # Both xs and ys_in are lists of arrays.
        exs = sequence_embed(self.embed_x, xs)
        eys = sequence_embed(self.embed_y, ys_in)

        batch = len(xs)
        # None represents a zero vector in an encoder.
        hx, cx, _ = self.encoder(None, None, exs)
        _, _, os = self.decoder(hx, cx, eys)

        # It is faster to concatenate data before calculating loss
        # because only one matrix multiplication is called.
        concat_os = F.concat(os, axis=0)
        concat_ys_out = F.concat(ys_out, axis=0)
        loss = F.sum(F.softmax_cross_entropy(
            self.W(concat_os), concat_ys_out, reduce='no')) / batch

        chainer.report({'loss': loss}, self)
        n_words = concat_ys_out.shape[0]
        perp = self.xp.exp(loss.array * batch / n_words)
        chainer.report({'perp': perp}, self)
        return loss

    def translate(self, xs, max_length=100):
        batch = len(xs)
        with chainer.no_backprop_mode(), chainer.using_config('train', False):
            xs = [x[::-1] for x in xs]
```

(continues on next page)

(continued from previous page)

```
        exs = sequence_embed(self.embed_x, xs)
        h, c, _ = self.encoder(None, None, exs)
        ys = self.xp.full(batch, EOS, numpy.int32)
        result = []
        for i in range(max_length):
            eys = self.embed_y(ys)
            eys = F.split_axis(eys, batch, 0)
            h, c, ys = self.decoder(h, c, eys)
            cys = F.concat(ys, axis=0)
            wy = self.W(cys)
            ys = self.xp.argmax(wy.array, axis=1).astype(numpy.int32)
            result.append(ys)

    # Using `xp.concatenate(...)` instead of `xp.stack(result)` here to
    # support NumPy 1.9.
    result = chainer.get_device('@numpy').send(
        self.xp.concatenate([x[None, :] for x in result]).T)

    # Remove EOS taggs
    outs = []
    for y in result:
        inds = numpy.argwhere(y == EOS)
        if len(inds) > 0:
            y = y[:inds[0, 0]]
        outs.append(y)
    return outs
```

- In `Seq2seq`, three functions are defined: the constructor `__init__`, the function call `forward`, and the function for translation `translate`.

Listing 34: seq2seq.py

```
def __init__(self, n_layers, n_source_vocab, n_target_vocab, n_units):
    super(Seq2seq, self).__init__()
    with self.init_scope():
        self.embed_x = L.EmbedID(n_source_vocab, n_units)
        self.embed_y = L.EmbedID(n_target_vocab, n_units)
        self.encoder = L.NStepLSTM(n_layers, n_units, n_units, 0.1)
        self.decoder = L.NStepLSTM(n_layers, n_units, n_units, 0.1)
        self.W = L.Linear(n_units, n_target_vocab)

    self.n_layers = n_layers
    self.n_units = n_units
```

- When we instantiate this class for making a model, we give the number of stacked lstms to `n_layers`, the vocabulary size of the source language to `n_source_vocab`, the vocabulary size of the target language to `n_target_vocab`, and the size of hidden vectors to `n_units`.
- This network uses *chainer.links.NStepLSTM*, *chainer.links.EmbedID*, and *chainer.links.Linear* as its building blocks. All the layers are registered and initialized in the context with `self.init_scope()`.
- You can access all the parameters in those layers by calling `self.params()`.
- In the constructor, it initializes all parameters with values sampled from a uniform distribution $U(-1, 1)$.

Listing 35: seq2seq.py

```
    def forward(self, xs, ys):
        xs = [x[::-1] for x in xs]

        eos = self.xp.array([EOS], numpy.int32)
        ys_in = [F.concat([eos, y], axis=0) for y in ys]
        ys_out = [F.concat([y, eos], axis=0) for y in ys]

        # Both xs and ys_in are lists of arrays.
        exs = sequence_embed(self.embed_x, xs)
        eys = sequence_embed(self.embed_y, ys_in)

        batch = len(xs)
        # None represents a zero vector in an encoder.
        hx, cx, _ = self.encoder(None, None, exs)
        _, _, os = self.decoder(hx, cx, eys)

        # It is faster to concatenate data before calculating loss
        # because only one matrix multiplication is called.
        concat_os = F.concat(os, axis=0)
        concat_ys_out = F.concat(ys_out, axis=0)
        loss = F.sum(F.softmax_cross_entropy(
            self.W(concat_os), concat_ys_out, reduce='no')) / batch

        chainer.report({'loss': loss}, self)
        n_words = concat_ys_out.shape[0]
        perp = self.xp.exp(loss.array * batch / n_words)
        chainer.report({'perp': perp}, self)
        return loss
```

- The `forward` method takes sequences of source language's word IDs `xs` and sequences of target language's word IDs `ys`. Each sequence represents a sentence, and the size of `xs` is mini-batch size.
- Note that the sequences of word IDs `xs` and `ys` are converted to a vocabulary-size one-hot vectors and then multiplied with the embedding matrix in `sequence_embed` to obtain embedding vectors `exs` and `eys`.

Listing 36: seq2seq.py

```
def sequence_embed(embed, xs):
    x_len = [len(x) for x in xs]
    x_section = numpy.cumsum(x_len[:-1])
    ex = embed(F.concat(xs, axis=0))
    exs = F.split_axis(ex, x_section, 0)
    return exs
```

- `self.encoder` and `self.decoder` are the encoder and the decoder of the seq2seq model. Each element of the decoder output `os` is $h^{(t)}_{[1:J]}$ in the figure above.
- After calculating the recurrent layer output, the loss `loss` and the perplexity `perp` are calculated, and the values are logged by *`chainer.report`*.

Note: It is well known that the seq2seq model learns much better when the source sentences are reversed. The paper[1] says that "*While the LSTM is capable of solving problems with long term dependencies, we discovered that the LSTM learns much better when the source sentences are reversed (the target sentences are not reversed). By doing so, the LSTM's test perplexity dropped from 5.8 to 4.7, and the test BLEU scores of its decoded translations increased from 25.9 to 30.6.*" So, at the first line in the `forward`, the input sentences are reversed `xs = [x[::-1] for x` in xs].

Listing 37: seq2seq.py

```
    def translate(self, xs, max_length=100):
        batch = len(xs)
        with chainer.no_backprop_mode(), chainer.using_config('train', False):
            xs = [x[::-1] for x in xs]
            exs = sequence_embed(self.embed_x, xs)
            h, c, _ = self.encoder(None, None, exs)
            ys = self.xp.full(batch, EOS, numpy.int32)
            result = []
            for i in range(max_length):
                eys = self.embed_y(ys)
                eys = F.split_axis(eys, batch, 0)
                h, c, ys = self.decoder(h, c, eys)
                cys = F.concat(ys, axis=0)
                wy = self.W(cys)
                ys = self.xp.argmax(wy.array, axis=1).astype(numpy.int32)
                result.append(ys)

        # Using `xp.concatenate(...)` instead of `xp.stack(result)` here to
        # support NumPy 1.9.
        result = chainer.get_device('@numpy').send(
            self.xp.concatenate([x[None, :] for x in result]).T)

        # Remove EOS taggs
        outs = []
        for y in result:
            inds = numpy.argwhere(y == EOS)
            if len(inds) > 0:
                y = y[:inds[0, 0]]
            outs.append(y)
        return outs
```

- After the model learned the parameters, the function `translate` is called to generate the translated sentences `outs` from the source sentences `xs`.
- So as not to change the parameters, the codes for the translation are nested in the scope `chainer.no_backprop_mode()` and `chainer.using_config('train', False)`.

2.2.4 Load French-English Corpus from WMT15 Dataset

In this tutorial, we use French-English corpus from WMT15 website that contains 10^9 documents. We must prepare additional libraries, dataset, and parallel corpus. To understand the pre-processing, see *2.3.1 Requirements*.

After the pre-processing the dataset, let's make dataset objects:

Listing 38: seq2seq.py

```
# Load pre-processed dataset
print('[{}] Loading dataset... (this may take several minutes)'.format(
    datetime.datetime.now()))
source_ids = load_vocabulary(args.SOURCE_VOCAB)
target_ids = load_vocabulary(args.TARGET_VOCAB)
```

(continues on next page)

(continued from previous page)

```
if args.use_dataset_api:
    # By using TextDataset, you can avoid loading whole dataset on memory.
    # This significantly reduces the host memory usage.
    def _filter_func(s, t):
        sl = len(s.strip().split())  # number of words in source line
        tl = len(t.strip().split())  # number of words in target line
        return (
            args.min_source_sentence <= sl <= args.max_source_sentence and
            args.min_target_sentence <= tl <= args.max_target_sentence)

    train_data = load_data_using_dataset_api(
        source_ids, args.SOURCE,
        target_ids, args.TARGET,
        _filter_func,
    )
else:
    # Load all records on memory.
    train_source = load_data(source_ids, args.SOURCE)
    train_target = load_data(target_ids, args.TARGET)
    assert len(train_source) == len(train_target)

    train_data = [
        (s, t)
        for s, t in six.moves.zip(train_source, train_target)
        if (args.min_source_sentence <= len(s) <= args.max_source_sentence
            and
            args.min_target_sentence <= len(t) <= args.max_target_sentence)
    ]
print('[{}] Dataset loaded.'.format(datetime.datetime.now()))

if not args.use_dataset_api:
    # Skip printing statistics when using TextDataset API, as it is slow.
    train_source_unknown = calculate_unknown_ratio(
        [s for s, _ in train_data])
    train_target_unknown = calculate_unknown_ratio(
        [t for _, t in train_data])

    print('Source vocabulary size: %d' % len(source_ids))
    print('Target vocabulary size: %d' % len(target_ids))
    print('Train data size: %d' % len(train_data))
    print('Train source unknown ratio: %.2f%%' % (
        train_source_unknown * 100))
    print('Train target unknown ratio: %.2f%%' % (
        train_target_unknown * 100))

target_words = {i: w for w, i in target_ids.items()}
source_words = {i: w for w, i in source_ids.items()}
```

- This code uses utility functions below:

Listing 39: seq2seq.py

```
def load_vocabulary(path):
    with io.open(path, encoding='utf-8') as f:
        # +2 for UNK and EOS
```

(continues on next page)

(continued from previous page)

```
        word_ids = {line.strip(): i + 2 for i, line in enumerate(f)}
    word_ids['<UNK>'] = 0
    word_ids['<EOS>'] = 1
    return word_ids
```

Listing 40: seq2seq.py

```
def load_data(vocabulary, path):
    n_lines = count_lines(path)
    bar = progressbar.ProgressBar()
    data = []
    print('loading...: %s' % path)
    with io.open(path, encoding='utf-8') as f:
        for line in bar(f, max_value=n_lines):
            words = line.strip().split()
            array = numpy.array([vocabulary.get(w, UNK)
                                 for w in words], numpy.int32)
            data.append(array)
    return data
```

Listing 41: seq2seq.py

```
def calculate_unknown_ratio(data):
    unknown = sum((s == UNK).sum() for s in data)
    total = sum(s.size for s in data)
    return unknown / total
```

2.2.5 Define Evaluation Function (Bleu Score)

BLEU[3] (bilingual evaluation understudy) is the evaluation metric for the quality of text which has been machine-translated from one natural language to another.

Listing 42: seq2seq.py

```
class CalculateBleu(chainer.training.Extension):

    trigger = 1, 'epoch'
    priority = chainer.training.PRIORITY_WRITER

    def __init__(
            self, model, test_data, key, device, batch=100, max_length=100):
        self.model = model
        self.test_data = test_data
        self.key = key
        self.batch = batch
        self.device = device
        self.max_length = max_length

    def __call__(self, trainer):
        device = self.device

        with chainer.no_backprop_mode():
            references = []
            hypotheses = []
```

(continues on next page)

(continued from previous page)

```
            for i in range(0, len(self.test_data), self.batch):
                sources, targets = zip(*self.test_data[i:i + self.batch])
                references.extend([[t.tolist()] for t in targets])

                sources = [device.send(x) for x in sources]
                ys = [y.tolist()
                      for y in self.model.translate(sources, self.max_length)]
                hypotheses.extend(ys)

        bleu = bleu_score.corpus_bleu(
            references, hypotheses,
            smoothing_function=bleu_score.SmoothingFunction().method1)
        chainer.report({self.key: bleu})
```

2.2.6 Create Iterator

Here, the code below just creates iterator objects.

Listing 43: seq2seq.py

```
train_iter = chainer.iterators.SerialIterator(train_data, args.batchsize)
```

2.2.7 Create RNN and Classification Model

Instantiate `Seq2seq` model.

Listing 44: seq2seq.py

```
model = Seq2seq(args.layer, len(source_ids), len(target_ids), args.unit)
```

2.2.8 Setup Optimizer

Prepare an optimizer. We use *chainer.optimizers.Adam*.

Listing 45: seq2seq.py

```
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)
```

2.2.9 Setup and Run Trainer

Let's make a trainer object.

Listing 46: seq2seq.py

```
updater = training.updaters.StandardUpdater(
    train_iter, optimizer, converter=convert, device=device)
```

(continues on next page)

(continued from previous page)

```
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)
trainer.extend(extensions.LogReport(
    trigger=(args.log_interval, 'iteration')))
trainer.extend(extensions.PrintReport(
    ['epoch', 'iteration', 'main/loss', 'main/perp',
     'validation/main/bleu', 'elapsed_time']),
    trigger=(args.log_interval, 'iteration'))

trainer.extend(
    extensions.snapshot(filename='snapshot_epoch_{.updater.iteration}'),
    trigger=(args.validation_interval, 'iteration'))
```

Setup the trainer's extension to see the BLEU score on the test data.

Listing 47: seq2seq.py

```
    test_source = load_data(source_ids, args.validation_source)
    test_target = load_data(target_ids, args.validation_target)
    assert len(test_source) == len(test_target)
    test_data = list(six.moves.zip(test_source, test_target))
    test_data = [(s, t) for s, t in test_data if 0 < len(s) and 0 < len(t)]
    test_source_unknown = calculate_unknown_ratio(
        [s for s, _ in test_data])
    test_target_unknown = calculate_unknown_ratio(
        [t for _, t in test_data])

    print('Validation data: %d' % len(test_data))
    print('Validation source unknown ratio: %.2f%%' %
          (test_source_unknown * 100))
    print('Validation target unknown ratio: %.2f%%' %
          (test_target_unknown * 100))

    @chainer.training.make_extension()
    def translate(trainer):
        source, target = test_data[numpy.random.choice(len(test_data))]
        result = model.translate([model.xp.array(source)])[0]

        source_sentence = ' '.join([source_words[x] for x in source])
        target_sentence = ' '.join([target_words[y] for y in target])
        result_sentence = ' '.join([target_words[y] for y in result])
        print('# source : ' + source_sentence)
        print('# result : ' + result_sentence)
        print('# expect : ' + target_sentence)

    trainer.extend(
        translate, trigger=(args.validation_interval, 'iteration'))
    trainer.extend(
        CalculateBleu(
            model, test_data, 'validation/main/bleu', device),
        trigger=(args.validation_interval, 'iteration'))

if args.resume is not None:
    # Resume from a snapshot
    chainer.serializers.load_npz(args.resume, trainer)
```

Let's start the training!

Listing 48: seq2seq.py

```
trainer.run()

if args.save is not None:
    # Save a snapshot
    chainer.serializers.save_npz(args.save, trainer)
```

2.3 Run Example

2.3.1 Requirements

Before running the example, you must prepare additional libraries, dataset, and parallel corpus.

- See the detail description: chainer/examples/seq2seq/README.md

2.3.1 Training the model

You can train the model with the script: chainer/examples/seq2seq/seq2seq.py

```
$ pwd
/root2chainer/chainer/examples/seq2seq
$ python seq2seq.py --gpu=0 giga-fren.preprocess.en giga-fren.preprocess.fr \
vocab.en vocab.fr \
--validation-source newstest2013.preprocess.en \
--validation-target newstest2013.preprocess.fr > log
100% (22520376 of 22520376) |#############| Elapsed Time: 0:09:20 Time: 0:09:20
100% (22520376 of 22520376) |#############| Elapsed Time: 0:10:36 Time: 0:10:36
100% (3000 of 3000) |#####################| Elapsed Time: 0:00:00 Time: 0:00:00
100% (3000 of 3000) |#####################| Elapsed Time: 0:00:00 Time: 0:00:00
epoch       iteration   main/loss   validation/main/loss  main/perp   validation/main/
↪perp  validation/main/bleu  elapsed_time
0           200         171.449                           991.556                    ␣
↪                             85.6739
0           400         143.918                           183.594                    ␣
↪                             172.473
0           600         133.48                            126.945                    ␣
↪                             260.315
0           800         128.734                           104.127                    ␣
↪                             348.062
0           1000        124.741                           91.5988                    ␣
↪                             436.536
...
```

Note: Before running the script, be careful the locale and the python's encoding. Please setup them to use utf-8 encoding.

2.3.1 Validate the model

While you are training the model, you can get the validation results:

```
...
# source : We knew the Government had tried many things , like launching <UNK> with
↪<UNK> or organising speed dating evenings .
# result : Nous savions que le gouvernement avait <UNK> plusieurs fois , comme le
↪<UNK> <UNK> , le <UNK> ou le <UNK> <UNK> .
# expect : Nous savions que le gouvernement avait tenté plusieurs choses comme lancer␣
↪des parfums aux <UNK> ou organiser des soirées de <UNK>
...
```

3.8.4 3. Reference

- [1] Sequence to Sequence Learning with Neural Networks
- [2] Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation
- [3] BLEU

CHAPTER

FOUR

API REFERENCE

4.1 Variable and Parameter

4.1.1 Variable classes and utilities

chainer.Variable	Array with a structure to keep track of computation.
chainer.as_array	Returns the underlying array from a variable or an array.
chainer.as_variable	Converts an array or a variable into *Variable*.
chainer.backward	Runs backpropagation from variables simultaneously.
chainer.Parameter	Parameter variable that can be registered to a link.
chainer.variable.VariableNode	Node in the backward computational graph representing a variable.

chainer.Variable

class chainer.**Variable**(*data=None*, ***, *name=None*, *grad=None*, *requires_grad=True*)

Array with a structure to keep track of computation.

Every variable holds a data array of type either `numpy.ndarray` or `cupy.ndarray`.

A variable object holds a data array and a *VariableNode* object of a computational graph. If the variable is constructed by the user, the node is *root* and does not hold any parent. If the variable is constructed by a *FunctionNode* object (i.e., by calling functions under `chainer.functions` or user-defined functions), or by using operators (see the list below), the node holds a reference to its parent called *creator_node*. This reference is used in backpropagation to backtrack the graph.

Users can disable (resp. enable) this chaining behavior by calling *no_backprop_mode()* (resp. *force_backprop_mode()*). In the former context, a variable never creates a computational graph, whereas in the latter context, it is forced to create.

Note: The following operators are defined for variable(s).

- Indexing: `a[slices]` (*__getitem__()*)
- Addition: `a + b` (*__add__()*, *__radd__()*)
- Subtraction: `a - b` (*__sub__()*, *__rsub__()*)
- Multiplication: `a * b` (*__mul__()*, *__rmul__()*)
- Division: `a / b` (*__div__()*, *__rdiv__()*, *__truediv__()*, *__rtruediv__()*)
- Floor Division: `a // b` (*__floordiv__()*, *__rfloordiv__()*)

- Exponentiation: `a ** b` (`__pow__()`, `__rpow__()`)
- Matrix Multiplication: `a @ b` (`__matmul__()`, `__rmatmul__()`)
- Negation (Arithmetic): `- a` (`__neg__()`)
- Absolute value: `abs(a)` (`__abs__()`)

Parameters

- **data** (*N-dimensional array*) – Initial data array.
- **name** (*str*) – Name of the variable.
- **grad** (*N-dimensional array*) – Initial gradient array.
- **requires_grad** (*bool*) – Boolean indicating whether `grad` will be set in backward calculation.

Methods

__getitem__(*slices*)

Extract elements from array with specified shape, axes and offsets.

Parameters

- **x** (`Variable` or *N-dimensional array*) – A variable to be sliced.
- **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – An object to specify the selection of elements.

Returns A `Variable` object which contains sliced array of `x`.

Note: It only supports types that are supported by CUDA's atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

Note: It does not support `slices` that contains multiple boolean arrays.

Note: See NumPy documentation for details of indexing.

Example

```
>>> x = np.arange(12).reshape((2, 2, 3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
>>> F.get_item(x, 0)
variable([[0, 1, 2],
```

(continues on next page)

(continued from previous page)

```
          [3,  4,  5]])
>>> F.get_item(x, (0, 0, slice(0, 2, 1)))  # equals x[0, 0, 0:2:1]
variable([0, 1])
>>> F.get_item(x, (Ellipsis, 2))  # equals x[..., 2]
variable([[ 2,  5],
          [ 8, 11]])
>>> F.get_item(x, (1, np.newaxis, 1, 0))  # equals x[1, None, 1, 0]
variable([9])
```

__len__()

Returns the first dimension of the data array.

Returns Number of the first dimension of the data array.

Return type int

__copy__()

addgrad(*var*)

Accumulates the gradient array from given source variable.

This method adds the gradient of a given variable to the gradient of this variable. The accumulation is even done across the host and different devices. If this variable has uninitialized data/grad arrays, this method initializes it with the shape of the given variable and then accumulates the gradient.

Parameters **var** (`Variable`) – Source variable.

backward(*retain_grad=False*, *enable_double_backprop=False*, *loss_scale=None*)

Runs error backpropagation (a.k.a. backprop) from this variable.

On backprop, *`FunctionNode.backward()`* is called on each *`FunctionNode`* object appearing in the backward graph starting from this variable. The backward graph is represented by backward references from variable nodes to their creators, and from function nodes to their input variable nodes. The backprop stops at all root nodes. Some function nodes set `None` as gradients of some inputs, where further backprop does not take place at such inputs.

This method uses *`grad`* as the initial error array. User can manually set a gradient array before calling this method. If the shape of *`data`* is `()` (i.e., it is scalar) and *`grad`* is `None`, then this method automatically complements 1.0 as the initial error. This is useful on starting backprop from some scalar loss value.

From v3, this method supports *differentiable backprop* (a.k.a. double backprop, grad of grads). To enable it, pass `enable_double_backprop=True`.

Parameters

- **retain_grad** (*`bool`*) – If `True`, the gradient arrays of all intermediate variables are kept. Otherwise, *`grad`* of the intermediate variables are set to `None` on appropriate timing, which may reduce the maximum memory consumption.

  In most cases of training some models, the purpose of backprop is to compute gradients of parameters, not of all variables, and therefore it is recommended that this flag be set to `False`.

- **enable_double_backprop** (*`bool`*) – *(Added in v3.0)* If `True`, computational trace of the whole backpropagation procedure is recorded to the computational graph so that one can further do backpropagation from the resulting gradients. Note that enabling it results in larger memory consumption needed to store the gradients w.r.t intermediate variables that are required for the second gradient computation.

- **loss_scale** (*float*) – Loss scaling factor. Loss scaling is a usefull technique to mitigate vanishing gradient issue that tends to happen when low precision data type like float16 is used during training. If you set loss scaling factor, gradients of loss values are to be multiplied by the factor before backprop starts. The factor is propagated to whole gradients in a computational graph along the backprop. The gradients of parameters are divided by the factor just before the parameters are to be updated.

cleargrad()
Clears the gradient array.

copydata(*var*)
Copies the data array from given source variable.

This method copies the data array from given variable to this variable. The copy is done even if the arrays reside on different devices, including across the host and a GPU device. If this variable has an uninitialized data array, this method initializes it by the data array of the given variable. Similarly, if the given variable has an uninitialized data array, this method initializes it by the data array of this variable (`self`). If both are uninitialized, this method does nothing.

Parameters **var** (`Variable`) – Source variable.

debug_print()
Display a summary of the stored data and location of the Variable

from_chx()
Converts the array and gradient to non-ChainerX arrays without copy.

This method converts the underlying ChainerX array and gradient residing in either a `native` or `cuda` device to NumPy or CuPy arrays respectively, on their same physical device. It does nothing if the array held by the Variable object is not a ChainerX array. The new array is a view of the original one.

Raises an error if such a conversion is not supported for the device.

item()
Converts the variable with one element to a Python scalar.

This will incur host-device synchronization.

Returns The element of the array.

Return type int or float

mean(*axis=None*, *weights=None*, *keepdims=False*)
Calculate weighted average of array elements over a given axis.

See also:

chainer.functions.average() for full documentation,

reshape(**shape*)
Returns a variable of a different shape and the same content.

See also:

chainer.functions.reshape() for full documentation,

retain_data()
Lets the corresponding variable node keep the underlying array.

set_creator(*gen_func*)
Notifies the variable that the given function is its creator.

Parameters **gen_func** (`Function`) – Function object that creates this variable as one of its outputs.

set_creator_node(*fnode*)
Notifies the variable that the given node is its creator.

Parameters **fnode** (`FunctionNode`) – Function node that has this variable as an output.

summary()

to_chx()
Converts the array and gradient to ChainerX arrays without copy.

This method converts the underlying array and gradient to *`chainerx.ndarray`* on the same physical device. It does nothing if the array held by the Variable object is already a ChainerX array. The new array is a view of the original one.

to_cpu()
Copies the data and gradient arrays to CPU.

to_device(*device*)
Copies the data and gradient arrays to specified device.

Parameters **device** – Target device specifier. See *`get_device()`* for available values.

to_gpu(*device=None*)
Copies the data and gradient arrays to specified GPU.

Parameters **device** – Target device specifier. If omitted, the current device is used.

to_intel64()
Copies the data and gradient arrays to intel64 specific mdarray.

If the array is not suited for intel64, it will be converted to `numpy.ndarray`.

transpose(**axes*)
Permute the dimensions of an input variable without copy.

See also:

`chainer.functions.transpose()` for full documentation.

unchain()
Deletes the reference to the creator of this variable.

This method deletes the reference to the creator from the corresponding variable node. Unlike *`unchain_backward()`*, it does not backtrack the graph.

This method is equivalent to `self.creator_node = None`.

unchain_backward()
Deletes references between variable nodes and functions backward.

After this method completes, intermediate variable nodes and functions that are not referenced from anywhere are deallocated by reference count GC. Also this variable itself deletes the reference to its creator function from the node, i.e. the node becomes root in the computation graph. It indicates that backprop after unchaining stops at this variable. This behavior is useful to implement truncated BPTT.

zerograd()
Initializes the gradient array by zeros.

Note that the gradient variable is unchained from the computational graph by this method, because this operation breaks the backprop validity.

Deprecated since version v1.15: Use more efficient `cleargrads()` instead.

__eq__(*other*)
This operator is not supported in Variables.

__ne__(*other*)
This operator is not supported in Variables.

__lt__(*other*)
This operator is not supported in Variables.

__le__(*other*)
This operator is not supported in Variables.

__gt__(*other*)
This operator is not supported in Variables.

__ge__(*other*)
This operator is not supported in Variables.

__nonzero__()
This operator is not supported in Variables.

__bool__()
This operator is not supported in Variables.

__neg__()
Element-wise negation.

Returns Output variable.

Return type *Variable*

__abs__()
Element-wise absolute.

Returns Output variable.

Return type *Variable*

__add__()
Element-wise addition.

Returns Output variable.

Return type *Variable*

__radd__()
Element-wise addition.

Returns Output variable.

Return type *Variable*

__sub__(*rhs*)
Element-wise subtraction.

Returns Output variable.

Return type *Variable*

__rsub__(*rhs*)
Element-wise subtraction.

Returns Output variable.

Return type *Variable*

__mul__(*rhs*)
Element-wise multiplication.

Returns Output variable.

Return type *Variable*

__rmul__(*rhs*)
Element-wise multiplication.

Returns Output variable.

Return type *Variable*

__div__(*rhs*)
Element-wise division

Returns Output variable.

Return type *Variable*

__truediv__(*rhs*)
Element-wise division

Returns Output variable.

Return type *Variable*

__rdiv__(*rhs*)
Element-wise division.

Returns Output variable.

Return type *Variable*

__rtruediv__(*rhs*)
Element-wise division.

Returns Output variable.

Return type *Variable*

__floordiv__(*rhs*)
Element-wise floor division.

Returns Output variable.

Return type *Variable*

__rfloordiv__(*rhs*)
Element-wise floor division.

Returns Output variable.

Return type *Variable*

__pow__(*rhs*)
Element-wise power function.

Returns Output variable.

Return type *Variable*

__rpow__(*rhs*)
Element-wise power function.

Returns Output variable.

Return type *Variable*

__matmul__(*rhs*)
Matrix multiplication.

Returns Output variable.

Return type *Variable*

__rmatmul__(*rhs*)
Matrix multiplication.

Returns Output variable.

Return type *Variable*

Attributes

T
Transposition of this variable.

array
The underlying data array.

It is either `numpy.ndarray` or `cupy.ndarray` object, or `None` if the variable in in an uninitialized state.

chx_array
A view of the raw ChainerX array.

In contrary to *`Variable.array`* which is always disconnected, the array represented by this attribute may be connected to the computational graph.

It is a view, so it has a distinct gradient from the original array.

If this attribute is queried on a *`Variable`* with a non-ChainerX array, `ValueError` will be raised.

creator
Function implementation that created this variable.

When this variable has been created by an old-style function (i.e., it is implemented as a subclass of *`Function`*), this property returns that *`Function`* object.

When this variable has been created by a new-style function (i.e., it is implemented as a subclass of *`FunctionNode`* class), this property returns that node object.

creator_node
`FunctionNode` object that created this variable.

This property has a setter to which `None` can be set. Setting `None` to this property is equivalent to call *`unchain()`*; it purges the variable from the function that created this variable.

The setter also accepts the original *`FunctionNode`* object that created this variable. For example, you can once set `None` to this property and then set the original value again.

Note: Setting an irrelevant *`FunctionNode()`* object does not emit any error immediately, whereas the behavior is undefined. Do not set a *`FunctionNode()`* object that did not create this variable object.

---

data
The underlying data array (equivalent to *`array`*).

Note that using this attribute directly is discouraged; use *`array`* instead. Using *`array`*, you can find an error earlier when your code mixes up Variable and ndarray because ndarray does not have an attribute `.array` while it has `.data`.

device
Device on which the data array of this variable reside.

dtype

grad
Gradient array of this variable.

Note that this property returns the underlying array of the gradient variable instead of the gradient variable itself; to get/set gradient variable, use `grad_var` instead.

If the underlying array is a `chainerx.ndarray` and requires_grad is false, trying to access the gradient will results in and error.

grad_var
Gradient variable.

label
Short text that represents the variable.

name

ndim

node

rank

requires_grad
It indicates that `grad` will be set in backward calculation.

shape

size

xp
Array module for the data array of this variable.

chainer.as_array

`chainer.`**`as_array`**`(`*obj*`)`
Returns the underlying array from a variable or an array.

This is a convenient function to get the underlying array object transparently from an object that could be either a variable or an array.

Parameters **obj** (*N-dimensional array* or ~chainer.Variable) – An array or a variable.

Returns The underlying array object of the argument.

Return type *N-dimensional array* or ~chainer.Variable

chainer.as_variable

`chainer.`**`as_variable`**`(`*obj*`)`
Converts an array or a variable into `Variable`.

This is a convenient function to get a `Variable` object transparently from a raw array or a variable.

Note that this function should only be used for type consistency (i.e., to enforce the return value of an API having type `Variable`). The `requires_grad` flag is kept as is; if `obj` is a raw array, the newly created variable has `requires_grad = False`. In order to make a variable w.r.t. which you want to compute the gradient, you should use `Variable` directly.

Parameters **obj** (*N-dimensional array* or ~chainer.Variable) – An array or a variable that you want to convert to `Variable`.

Returns A variable converted from `obj`. If `obj` is a raw array, this is a new `Variable` object that wraps the array. If `obj` is already a `Variable` object, this function returns `obj` as is.

Return type *Variable*

chainer.backward

`chainer.backward(outputs, grad_outputs=None, *, enable_double_backprop=False)`

Runs backpropagation from variables simultaneously.

> **Warning:** This feature is experimental. The interface can change in the future.

Parameters

- **outputs** (tuple or list of `Variable`) – A sequence of output variables from which backprop starts.
- **grad_outputs** (None or tuple or list of `Variable`) – A sequence of variables that gives the initial value of each output gradient. If this argument is `None`, backprop uses `grad_var` of `outputs`.
- **enable_double_backprop** (`bool`) – If `True`, computational trace of the whole backpropagation procedure is recorded to the computational graph so that one can further do backpropagation from the resulting gradients. Note that enabling it results in larger memory consumption needed to store the gradients w.r.t intermediate variables that are required for the second gradient computation.

See also:

`chainer.Variable.backward()` `chainer.grad()`

chainer.Parameter

`class chainer.Parameter(initializer=None, shape=None, name=None)`

Parameter variable that can be registered to a link.

Parameter is a subclass of `Variable`. It almost behaves as same as a usual variable except that a parameter can be registered to a `Link` object just by assigning it to an attribute of the link within an `init_scope()` context.

Parameter also supports an initialization by an initializer. It can have two initializers: one for the data array, and the other for the gradient array. The initializer only specifies the way of filling the elements of these arrays, and the shape information is specified at the initialization point.

When a link that the parameter has been registered to is passed to an `GradientMethod`, an update rule is set to the parameter. This update rule specifies how to update the data array of the parameter using its gradient array.

Parameters

- **initializer** (~chainer.Initializer or *N-dimensional array*) – Initializer of the data array. If `shape` is given, this initializer is immediately used to initialize the data array. Otherwise, if it is an array, it is immediately used as the data array, and otherwise the data array is left uninitialized and will be initialized by this initializer in `initialize()`. It can also be a

scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

- **shape** (*int or tuple of int or None*) – Shape of the parameter. If it is `None`, the initialization is deferred to the call of `initialize()`.
- **name** (*str*) – Name of the parameter.

Variables

- ***initializer*** – Initializer of the data array. It is used for initializing the data array of an uninitialized variable.
- **update_rule** – `UpdateRule` instance that updates this variable as a parameter. This argument is set to `update_rule`.

Methods

__getitem__(*slices*)

Extract elements from array with specified shape, axes and offsets.

Parameters

- **x** (*Variable* or *N-dimensional array*) – A variable to be sliced.
- **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – An object to specify the selection of elements.

Returns A *Variable* object which contains sliced array of `x`.

Note: It only supports types that are supported by CUDA's atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

Note: It does not support `slices` that contains multiple boolean arrays.

Note: See NumPy documentation for details of indexing.

Example

```
>>> x = np.arange(12).reshape((2, 2, 3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
>>> F.get_item(x, 0)
variable([[0, 1, 2],
          [3, 4, 5]])
>>> F.get_item(x, (0, 0, slice(0, 2, 1)))  # equals x[0, 0, 0:2:1]
variable([0, 1])
```

(continues on next page)

(continued from previous page)

```
>>> F.get_item(x, (Ellipsis, 2))  # equals x[..., 2]
variable([[ 2,  5],
          [ 8, 11]])
>>> F.get_item(x, (1, np.newaxis, 1, 0))  # equals x[1, None, 1, 0]
variable([9])
```

__len__()

Returns the first dimension of the data array.

Returns Number of the first dimension of the data array.

Return type int

__copy__()

addgrad(*var*)

Accumulates the gradient array from given source variable.

This method adds the gradient of a given variable to the gradient of this variable. The accumulation is even done across the host and different devices. If this variable has uninitialized data/grad arrays, this method initializes it with the shape of the given variable and then accumulates the gradient.

Parameters **var** (`Variable`) – Source variable.

backward(*retain_grad=False*, *enable_double_backprop=False*, *loss_scale=None*)

Runs error backpropagation (a.k.a. backprop) from this variable.

On backprop, `FunctionNode.backward()` is called on each `FunctionNode` object appearing in the backward graph starting from this variable. The backward graph is represented by backward references from variable nodes to their creators, and from function nodes to their input variable nodes. The backprop stops at all root nodes. Some function nodes set `None` as gradients of some inputs, where further backprop does not take place at such inputs.

This method uses `grad` as the initial error array. User can manually set a gradient array before calling this method. If the shape of `data` is `()` (i.e., it is scalar) and `grad` is `None`, then this method automatically complements 1.0 as the initial error. This is useful on starting backprop from some scalar loss value.

From v3, this method supports *differentiable backprop* (a.k.a. double backprop, grad of grads). To enable it, pass `enable_double_backprop=True`.

Parameters

- **retain_grad** (`bool`) – If `True`, the gradient arrays of all intermediate variables are kept. Otherwise, `grad` of the intermediate variables are set to `None` on appropriate timing, which may reduce the maximum memory consumption.

  In most cases of training some models, the purpose of backprop is to compute gradients of parameters, not of all variables, and therefore it is recommended that this flag be set to `False`.

- **enable_double_backprop** (`bool`) – *(Added in v3.0)* If `True`, computational trace of the whole backpropagation procedure is recorded to the computational graph so that one can further do backpropagation from the resulting gradients. Note that enabling it results in larger memory consumption needed to store the gradients w.r.t intermediate variables that are required for the second gradient computation.

- **loss_scale** (`float`) – Loss scaling factor. Loss scaling is a usefull technique to mitigate vanishing gradient issue that tends to happen when low precision data type like float16 is used during training. If you set loss scaling factor, gradients of loss values are to be multiplied by the factor before backprop starts. The factor is propagated to whole gradients in a computational graph along the backprop. The gradients of parameters are divided by the factor just before the parameters are to be updated.

cleargrad()
Clears the gradient array.

copydata(*var*)
Copies the data array from given source variable.

This method copies the data array from given variable to this variable. The copy is done even if the arrays reside on different devices, including across the host and a GPU device. If this variable has an uninitialized data array, this method initializes it by the data array of the given variable. Similarly, if the given variable has an uninitialized data array, this method initializes it by the data array of this variable (`self`). If both are uninitialized, this method does nothing.

Parameters **var** (`Variable`) – Source variable.

debug_print()
Display a summary of the stored data and location of the Variable

from_chx()
Converts the array and gradient to non-ChainerX arrays without copy.

This method converts the underlying ChainerX array and gradient residing in either a `native` or `cuda` device to NumPy or CuPy arrays respectively, on their same physical device. It does nothing if the array held by the Variable object is not a ChainerX array. The new array is a view of the original one.

Raises an error if such a conversion is not supported for the device.

initialize(*shape*)
Initializes the uninitialized variable.

Uninitialized variable is a variable created with the data array set to None. This method creates and initializes the data array. The shape of the variable can be left unknown until this method is called.

Parameters **shape** (*tuple of int*) – Shape of the data array.

item()
Converts the variable with one element to a Python scalar.

This will incur host-device synchronization.

Returns The element of the array.

Return type int or float

mean(*axis=None*, *weights=None*, *keepdims=False*)
Calculate weighted average of array elements over a given axis.

See also:

chainer.functions.average() for full documentation,

reshape(**shape*)
Returns a variable of a different shape and the same content.

See also:

chainer.functions.reshape() for full documentation,

retain_data()
Lets the corresponding variable node keep the underlying array.

set_creator(*gen_func*)
Notifies the variable that the given function is its creator.

Parameters **gen_func** (Function) – Function object that creates this variable as one of its outputs.

set_creator_node(*fnode*)
Notifies the variable that the given node is its creator.

Parameters **fnode** (FunctionNode) – Function node that has this variable as an output.

summary()

to_chx()
Converts the array and gradient to ChainerX arrays without copy.

This method converts the underlying array and gradient to *chainerx.ndarray* on the same physical device. It does nothing if the array held by the Variable object is already a ChainerX array. The new array is a view of the original one.

to_cpu()
Copies the data and gradient arrays to CPU.

to_device(*device*)
Copies the data and gradient arrays to specified device.

Parameters **device** – Target device specifier. See *get_device()* for available values.

to_gpu(*device=None*)
Copies the data and gradient arrays to specified GPU.

Parameters **device** – Target device specifier. If omitted, the current device is used.

to_intel64()
Copies the data and gradient arrays to intel64 specific mdarray.

If the array is not suited for intel64, it will be converted to `numpy.ndarray`.

transpose(**axes*)
Permute the dimensions of an input variable without copy.

See also:

chainer.functions.transpose() for full documentation.

unchain()
Deletes the reference to the creator of this variable.

This method deletes the reference to the creator from the corresponding variable node. Unlike *unchain_backward()*, it does not backtrack the graph.

This method is equivalent to `self.creator_node = None`.

unchain_backward()
Deletes references between variable nodes and functions backward.

After this method completes, intermediate variable nodes and functions that are not referenced from anywhere are deallocated by reference count GC. Also this variable itself deletes the reference to its creator function from the node, i.e. the node becomes root in the computation graph. It indicates that backprop after unchaining stops at this variable. This behavior is useful to implement truncated BPTT.

update()
Updates the data array using the gradient and the update rule.

This method updates the parameter using the attached update rule.

zerograd()
Initializes the gradient array by zeros.

Note that the gradient variable is unchained from the computational graph by this method, because this operation breaks the backprop validity.

Deprecated since version v1.15: Use more efficient `cleargrads()` instead.

__eq__(*other*)
This operator is not supported in Variables.

__ne__(*other*)
This operator is not supported in Variables.

__lt__(*other*)
This operator is not supported in Variables.

__le__(*other*)
This operator is not supported in Variables.

__gt__(*other*)
This operator is not supported in Variables.

__ge__(*other*)
This operator is not supported in Variables.

__nonzero__()
This operator is not supported in Variables.

__bool__()
This operator is not supported in Variables.

__neg__()
Element-wise negation.

Returns Output variable.

Return type *Variable*

__abs__()
Element-wise absolute.

Returns Output variable.

Return type *Variable*

__add__()
Element-wise addition.

Returns Output variable.

Return type *Variable*

__radd__()
Element-wise addition.

Returns Output variable.

Return type *Variable*

__sub__(*rhs*)
Element-wise subtraction.

Returns Output variable.

Return type *Variable*

__rsub__(*rhs*)
Element-wise subtraction.

Returns Output variable.

Return type *Variable*

`__mul__`(*rhs*)
Element-wise multiplication.

Returns Output variable.

Return type *Variable*

`__rmul__`(*rhs*)
Element-wise multiplication.

Returns Output variable.

Return type *Variable*

`__div__`(*rhs*)
Element-wise division

Returns Output variable.

Return type *Variable*

`__truediv__`(*rhs*)
Element-wise division

Returns Output variable.

Return type *Variable*

`__rdiv__`(*rhs*)
Element-wise division.

Returns Output variable.

Return type *Variable*

`__rtruediv__`(*rhs*)
Element-wise division.

Returns Output variable.

Return type *Variable*

`__floordiv__`(*rhs*)
Element-wise floor division.

Returns Output variable.

Return type *Variable*

`__rfloordiv__`(*rhs*)
Element-wise floor division.

Returns Output variable.

Return type *Variable*

`__pow__`(*rhs*)
Element-wise power function.

Returns Output variable.

Return type *Variable*

__rpow__(*rhs*)
Element-wise power function.

Returns Output variable.

Return type *Variable*

__matmul__(*rhs*)
Matrix multiplication.

Returns Output variable.

Return type *Variable*

__rmatmul__(*rhs*)
Matrix multiplication.

Returns Output variable.

Return type *Variable*

Attributes

T
Transposition of this variable.

array
The underlying data array.

It is either `numpy.ndarray` or `cupy.ndarray` object, or `None` if the variable in in an uninitialized state.

chx_array
A view of the raw ChainerX array.

In contrary to `Variable.array` which is always disconnected, the array represented by this attribute may be connected to the computational graph.

It is a view, so it has a distinct gradient from the original array.

If this attribute is queried on a `Variable` with a non-ChainerX array, `ValueError` will be raised.

creator
Function implementation that created this variable.

When this variable has been created by an old-style function (i.e., it is implemented as a subclass of `Function`), this property returns that `Function` object.

When this variable has been created by a new-style function (i.e., it is implemented as a subclass of `FunctionNode` class), this property returns that node object.

creator_node
`FunctionNode` object that created this variable.

This property has a setter to which `None` can be set. Setting `None` to this property is equivalent to call `unchain()`; it purges the variable from the function that created this variable.

The setter also accepts the original `FunctionNode` object that created this variable. For example, you can once set `None` to this property and then set the original value again.

Note: Setting an irrelevant `FunctionNode()` object does not emit any error immediately, whereas the behavior is undefined. Do not set a `FunctionNode()` object that did not create this variable object.

data
The underlying data array (equivalent to `array`).

Note that using this attribute directly is discouraged; use `array` instead. Using `array`, you can find an error earlier when your code mixes up Variable and ndarray because ndarray does not have an attribute `.array` while it has `.data`.

device
Device on which the data array of this variable reside.

dtype

grad
Gradient array of this variable.

Note that this property returns the underlying array of the gradient variable instead of the gradient variable itself; to get/set gradient variable, use `grad_var` instead.

If the underlying array is a `chainerx.ndarray` and requires_grad is false, trying to access the gradient will results in and error.

grad_var
Gradient variable.

initializer = None

label
Short text that represents the variable.

name

ndim

node

rank

requires_grad
It indicates that `grad` will be set in backward calculation.

shape

size

xp
Array module for the data array of this variable.

chainer.variable.VariableNode

class `chainer.variable.`**VariableNode**(*variable*, *name*, ***kwargs*)
Node in the backward computational graph representing a variable.

This object represents a variable node in a computational graph. The node is used in error backpropagation (a.k.a. backprop) to determine which gradient to be passed to each function.

A variable node is held by the corresponding `Variable` object, which is managed by users. `FunctionNode` objects that take the variable as an input also hold references to the variable node.

Note that the node does not hold a reference to the corresponding data array in general. The data array is actually accessible by the node in the following cases.

1. If there exists a `Variable` object that holds a reference to the variable node, the variable node holds a weak reference to the variable object, and thus the data array is accessible via the weak reference.

2. If `retain_data()` is called, the node holds a reference to the data array. It is mainly called by a function that needs the input or output data array in its backprop procedure. See `FunctionNode.retain_inputs()` and `FunctionNode.retain_outputs()` for more details.

Users usually do not need to touch this variable node object. The computational graph is automatically managed by Chainer, and any interface that is beneficial for users is also provided by `Variable`.

Parameters

- **variable** (`Variable`) – The corresponding variable object.
- **name** (`str`) – Name of the variable node.

Variables

- **dtype** – Data type of the data array.
- **shape** – Shape of the data array.
- **name** (`str`) – Name of the variable node.

Methods

get_variable()

Returns the corresponding `Variable` object.

VariableNode object holds a weak reference of the variable object. If the reference is alive, it is returned by this property. Otherwise, this property creates a new `Variable` object from this node object and returns it.

Returns The variable object that refers this node.

Return type *Variable*

get_variable_or_none()

Returns the holding `Variable` object or `None`.

VariableNode object holds a weak reference of the variable object.If the reference is alive, it is returned by this property. Otherwise, returns `None`.

Returns The variable object that refers this node.

Return type *Variable*

retain_data()

Lets the node hold a reference to the underlying data array.

This method gets the data array of the corresponding variable and keeps it. If the weak reference to the corresponding variable is dead, it raises an error.

set_creator(*creator*)

Sets a `Function` object that created this node.

This method is equivalent to `self.creator = creator`. A `FunctionNode` object can also be passed.

Parameters creator (`Function or FunctionNode`) – Function that has created this variable.

set_creator_node(*creator_node*)

Sets a `FunctionNode` object that created this node.

This method is equivalent to `self.creator_node = creator_node`. A `Function` object can also be passed, in which case the `Function.node` attribute is used.

Parameters **creator_node** (`FunctionNode` *or* `Function`) – Function node that has this variable as an output.

unchain()

Deletes the reference to the creator of this variable node.

This method is equivalent to `self.creator_node = None`.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

creator

Function object that created this variable node.

When the function is implemented with the old-style API (i.e., it uses `Function` class), this property returns the `Function` object. The object is extracted from the `FunctionAdapter` object, so the returned object is not the function node, but instead the actual implementation of forward and backward procedures.

When the function is implemented with the new-style API (i.e., it uses `FunctionNode` class), this property returns the function node object. In this case, the returned object is same as `creator_node`.

> **Warning:** As of v3.0.0, when the creator is an old-style function, the following code is invalid:
>
> ```
> creator = v.creator
> v.creator = None
> ...
> v.creator = creator
> ```
>
> The point is that `FunctionNode` objects are used as nodes in the computational graph instead of `Function`, and each `Function` object only holds a *weak reference* to the corresponding `FunctionNode`. Since `creator` returns the `Function` object, the `FunctionNode` object is not kept by preserving `creator`.
>
> The above code should be fixed as follows.
>
> ```
> creator_node = v.creator_node
> v.creator_node = None
> ...
> v.creator_node = creator_node
> ```

creator_node
Function node that has this variable as an output.

See *FunctionNode* for the definition of a function node.

data
Data array of the corresponding variable.

If the data is not available, it returns `None`.

grad
Gradient array of the corresponding variable.

If the variable is not available, it returns `None`.

grad_var
Gradient variable of the corresponding variable.

If the corresponding variable is not available, it return `None`.

label
Short text that represents the variable node.

rank

requires_grad
It indicates that `grad` will be set in backward calculation.

4.1.2 N-dimensional array

chainer.Variable holds its value as an n-dimensional array (ndarray). Chainer supports the following classes:

- `numpy.ndarray`, including `ideep4py.mdarray`
- `cupy.ndarray`
- *chainerx.ndarray*

Note: Python scalars (`float`, etc.) and NumPy scalars (`numpy.float16`, `numpy.float32`, etc.) cannot be used as *chainer.Variable.array*. See also `chainer.utils.force_array()`.

4.2 Functions

Chainer provides variety of built-in function implementations in *chainer.functions* package. These functions usually return a *Variable* object or a tuple of multiple *Variable* objects. For a *Variable* argument of a function, an *N-dimensional array* can be passed if you do not need its gradient. Some functions additionally supports scalar arguments.

Note: Functions implemented in Chainer consists of the following two parts:

- A class that inherits *FunctionNode*, which defines forward/backward computation.
- A "wrapper" function around the class.

APIs listed in this page are "wrapper" of *FunctionNode* implementations. In most cases, you don't have to use *FunctionNode* classes directly.

For example, `chainer.functions.sum()` is a wrapper function defined as `def sum(...):` in chainer/functions/math/sum.py, and it calls its corresponding `FunctionNode` implementation, `Sum`. Some functions may not have the corresponding `FunctionNode` implementation; one example is `chainer.functions.average()`, which is defined in chainer/functions/math/average.py, which calls other wrapper functions to calculate average.

If you are implementing your own functions, please see *Define your own function*.

4.2.1 Arithmetic functions

Basic arithmetic operations for `Variable`s are implemented as operators. Refer to the Notes section of `Variable` for details.

`chainer.functions.add()` provides better performance when accumulating three or more `Variable`s at once.

`chainer.functions.add`	Element-wise addition.

chainer.functions.add

`chainer.functions.add(*xs)`
Element-wise addition.

Returns Output variable.

Return type *Variable*

4.2.2 Activation functions

`chainer.functions.clipped_relu`	Clipped Rectifier Unit function.
`chainer.functions.crelu`	Concatenated Rectified Linear Unit function.
`chainer.functions.elu`	Exponential Linear Unit function.
`chainer.functions.hard_sigmoid`	Element-wise hard-sigmoid function.
`chainer.functions.leaky_relu`	Leaky Rectified Linear Unit function.
`chainer.functions.log_softmax`	Channel-wise log-softmax function.
`chainer.functions.lstm`	Long Short-Term Memory units as an activation function.
`chainer.functions.maxout`	Maxout activation function.
`chainer.functions.prelu`	Parametric ReLU function.
`chainer.functions.rrelu`	Randomized Leaky Rectified Liner Unit function.
`chainer.functions.relu`	Rectified Linear Unit function.
`chainer.functions.relu6`	Rectifier Unit function clipped at 6.
`chainer.functions.selu`	Scaled Exponential Linear Unit function.
`chainer.functions.sigmoid`	Element-wise sigmoid logistic function.
`chainer.functions.slstm`	S-LSTM units as an activation function.
`chainer.functions.softmax`	Softmax function.
`chainer.functions.softplus`	Element-wise softplus function.
`chainer.functions.swish`	Swish activation function.
`chainer.functions.tanh`	Elementwise hyperbolic tangent function.
`chainer.functions.tree_lstm`	TreeLSTM unit as an activation function.

chainer.functions.clipped_relu

`chainer.functions.clipped_relu(x, z=20.0)`

Clipped Rectifier Unit function.

For a clipping value $z(> 0)$, it computes

$$\text{ClippedReLU}(x, z) = \min(\max(0, x), z).$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_n)$-shaped float array.
- **z** (*float*) – Clipping value. (default = 20.0)

Returns Output variable. A $(s_1, s_2, ..., s_n)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.random.uniform(-100, 100, (10, 20)).astype(np.float32)
>>> z = 10.0
>>> np.any(x < 0)
True
>>> np.any(x > z)
True
>>> y = F.clipped_relu(x, z=z)
>>> np.any(y.array < 0)
False
>>> np.any(y.array > z)
False
```

chainer.functions.crelu

`chainer.functions.crelu(x, axis=1)`

Concatenated Rectified Linear Unit function.

This function is expressed as follows

$$f(x) = (\max(0, x), \max(0, -x)).$$

Here, two output values are concatenated along an axis.

See: https://arxiv.org/abs/1603.05201

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **axis** (*int*) – Axis that the output values are concatenated along. Default is 1.

Returns Output variable of concatenated array. If the axis is 1, A $(s_1, s_2 \times 2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.array([[-1, 0], [2, -3]], np.float32)
>>> x
array([[-1.,  0.],
       [ 2., -3.]], dtype=float32)
>>> y = F.crelu(x, axis=1)
>>> y.array
array([[0., 0., 1., 0.],
       [2., 0., 0., 3.]], dtype=float32)
```

---

chainer.functions.elu

`chainer.functions.elu(x, alpha=1.0)`

Exponential Linear Unit function.

For a parameter α, it is expressed as

$$f(x) = \left\{ \begin{array}{ll} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0, \end{array} \right.$$

See: https://arxiv.org/abs/1511.07289

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **alpha** (`float`) – Parameter α. Default is 1.0.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.array([[-1, 0], [2, -3]], np.float32)
>>> x
array([[-1.,  0.],
       [ 2., -3.]], dtype=float32)
>>> y = F.elu(x, alpha=1.)
>>> y.array
array([[-0.63212055,  0.        ],
       [ 2.        , -0.95021296]], dtype=float32)
```

---

chainer.functions.hard_sigmoid

`chainer.functions.hard_sigmoid(x)`

Element-wise hard-sigmoid function.

This function is defined as

$$f(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 0.2x + 0.5 & \text{if } -2.5 < x < 2.5 \\ 1 & \text{if } 2.5 < x. \end{cases}$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

It maps the input values into the range of $[0, 1]$.

```
>>> x = np.array([-2.6, -1, 0, 1, 2.6])
>>> x
array([-2.6, -1. ,  0. ,  1. ,  2.6])
>>> F.hard_sigmoid(x).array
array([0. , 0.3, 0.5, 0.7, 1. ])
```

---

chainer.functions.leaky_relu

`chainer.functions.leaky_relu(x, slope=0.2)`

Leaky Rectified Linear Unit function.

This function is expressed as

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0, \end{cases}$$

where a is a configurable slope value.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **slope** (`float`) – Slope value a.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.array([[-1, 0], [2, -3], [-2, 1]], np.float32)
>>> x
array([[-1.,  0.],
       [ 2., -3.],
       [-2.,  1.]], dtype=float32)
>>> F.leaky_relu(x, slope=0.2).array
array([[-0.2,  0. ],
       [ 2. , -0.6],
       [-0.4,  1. ]], dtype=float32)
```

---

chainer.functions.log_softmax

`chainer.functions.log_softmax(x, axis=1)`

Channel-wise log-softmax function.

This function computes its logarithm of softmax along the second axis. Let $c = (c_1, c_2, \dots, c_D)$ be the slice of `x` along with the second axis. For each slice c, it computes the logarithm of the function $f(c)$ defined as

$$f(c) = \frac{\exp(c)}{\sum_d \exp(c_d)}.$$

This method is theoretically equivalent to `log(softmax(x))` but is more stable.

Note: `log(softmax(x))` may cause underflow when `x` is too small, because `softmax(x)` may returns `0`. `log_softmax` method is more stable.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A n-dimensional ($n \geq 2$) float array.
- **axis** (*int*) – The axis along which the softmax is to be computed.

Returns Output variable. A n-dimensional ($n \geq 2$) float array, which is the same shape with x.

Return type *Variable*

See also:

softmax()

Example

```
>>> x = np.array([[0, 1, 2], [0, 2, 4]], np.float32)
>>> x
array([[0., 1., 2.],
       [0., 2., 4.]], dtype=float32)
>>> F.log_softmax(x).array
array([[-2.407606  , -1.4076059 , -0.4076059 ],
       [-4.1429315 , -2.1429315 , -0.14293146]], dtype=float32)
>>> np.allclose(F.log_softmax(x).data, F.log(F.softmax(x)).data)
True
```

chainer.functions.lstm

`chainer.functions.lstm(c_prev, x)`

Long Short-Term Memory units as an activation function.

This function implements LSTM units with forget gates. Let the previous cell state `c_prev` and the input array `x`.

First, the input array `x` is split into four arrays a, i, f, o of the same shapes along the second axis. It means that `x` 's second axis must have 4 times the `c_prev` 's second axis.

The split input arrays are corresponding to:

- a : sources of cell input
- i : sources of input gate
- f : sources of forget gate
- o : sources of output gate

Second, it computes the updated cell state `c` and the outgoing signal `h` as:

$$c = \tanh(a)\sigma(i) + c_{\text{prev}}\sigma(f),$$
$$h = \tanh(c)\sigma(o),$$

where σ is the elementwise sigmoid function. These are returned as a tuple of two variables.

This function supports variable length inputs. The mini-batch size of the current input must be equal to or smaller than that of the previous one. When mini-batch size of `x` is smaller than that of `c`, this function only updates `c[0:len(x)]` and doesn't change the rest of `c`, `c[len(x):]`. So, please sort input sequences in descending order of lengths before applying the function.

Parameters

- **c_prev** (*Variable* or *N-dimensional array*) – Variable that holds the previous cell state. The cell state should be a zero array or the output of the previous call of LSTM.
- **x** (*Variable* or *N-dimensional array*) – Variable that holds the sources of cell input, input gate, forget gate and output gate. It must have the second dimension whose size is four times of that of the cell state.

Returns Two *Variable* objects `c` and `h`. `c` is the updated cell state. `h` indicates the outgoing signal.

Return type tuple

See the original paper proposing LSTM with forget gates: Long Short-Term Memory in Recurrent Neural Networks.

See also:

LSTM

Example

Assuming `y` is the current incoming signal, `c` is the previous cell state, and `h` is the previous outgoing signal from an `lstm` function. Each of `y`, `c` and `h` has `n_units` channels. Most typical preparation of `x` is:

```
>>> n_units = 100
>>> y = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> h = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> c = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> model = chainer.Chain()
>>> with model.init_scope():
...   model.w = L.Linear(n_units, 4 * n_units)
...   model.v = L.Linear(n_units, 4 * n_units)
>>> x = model.w(y) + model.v(h)
>>> c, h = F.lstm(c, x)
```

It corresponds to calculate the input array `x`, or the input sources a, i, f, o, from the current incoming signal `y` and the previous outgoing signal `h`. Different parameters are used for different kind of input sources.

Note: We use the naming rule below.

- **incoming signal** The formal input of the formulation of LSTM (e.g. in NLP, word vector or output of lower RNN layer). The input of `chainer.links.LSTM` is the *incoming signal*.
- **input array** The array which is linear transformed from *incoming signal* and the previous outgoing signal. The *input array* contains four sources, the sources of cell input, input gate, forget gate and output gate. The input of `chainer.functions.activation.lstm.LSTM` is the *input array*.

chainer.functions.maxout

`chainer.functions.maxout(x, pool_size, axis=1)`

Maxout activation function.

It accepts an input tensor `x`, reshapes the `axis` dimension (say the size being `M * pool_size`) into two dimensions `(M, pool_size)`, and takes maximum along the `axis` dimension.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable. A n-dimensional ($n \geq$ `axis`) float array. In general, its first dimension is assumed to be the *minibatch dimension*. The other dimensions are treated as one concatenated dimension.
- **pool_size** (`int`) – The size used for downsampling of pooling layer.
- **axis** (`int`) – The `axis` dimension to be reshaped. The size of `axis` dimension should be `M * pool_size`.

Returns Output variable. The shape of the output is same as `x` except that `axis` dimension is transformed from `M * pool_size` to `M`.

Return type *Variable*

See also:

`Maxout`

Example

Typically, `x` is the output of a linear layer or a convolution layer. The following is the example where we use `maxout()` in combination with a Linear link.

```
>>> in_size, out_size, pool_size = 10, 10, 10
>>> bias = np.arange(out_size * pool_size).astype(np.float32)
>>> l = L.Linear(in_size, out_size * pool_size, initial_bias=bias)
>>> x = np.zeros((1, in_size), np.float32)  # prepare data
>>> x = l(x)
>>> y = F.maxout(x, pool_size)
>>> x.shape
(1, 100)
>>> y.shape
(1, 10)
>>> x.reshape((out_size, pool_size)).array
array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14., 15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24., 25., 26., 27., 28., 29.],
       [30., 31., 32., 33., 34., 35., 36., 37., 38., 39.],
       [40., 41., 42., 43., 44., 45., 46., 47., 48., 49.],
       [50., 51., 52., 53., 54., 55., 56., 57., 58., 59.],
       [60., 61., 62., 63., 64., 65., 66., 67., 68., 69.],
```

(continues on next page)

(continued from previous page)

```
       [70., 71., 72., 73., 74., 75., 76., 77., 78., 79.],
       [80., 81., 82., 83., 84., 85., 86., 87., 88., 89.],
       [90., 91., 92., 93., 94., 95., 96., 97., 98., 99.]], dtype=float32)
>>> y.array
array([[ 9., 19., 29., 39., 49., 59., 69., 79., 89., 99.]], dtype=float32)
```

chainer.functions.prelu

`chainer.functions.prelu(x, W)`

Parametric ReLU function.

It accepts two arguments: an input `x` and a weight array `W` and computes the output as

$$PReLU(x_i) = \begin{cases} x_i & (x_i > 0) \\ W_i * x_i & (otherwise) \end{cases}$$

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable. Its first axis is assumed to be the minibatch dimension.
- **W** (`Variable` or *N-dimensional array*) – Weight variable.

Returns Output variable

Return type *Variable*

Example

```
>>> x = np.arange(-3, 3, dtype=np.float32).reshape((2, 3))
>>> x
array([[-3., -2., -1.],
       [ 0.,  1.,  2.]], dtype=float32)
>>> W = np.array([0.01, 0.1, 1], dtype=np.float32)
>>> W
array([0.01, 0.1 , 1.  ], dtype=float32)
>>> F.prelu(x, W)
variable([[-0.03, -0.2 , -1.  ],
          [ 0.  ,  1.  ,  2.  ]])
```

Note: When the PReLU function is combined with two-dimensional convolution, the elements of parameter W are typically shared across the same filter of different pixels. In order to support such usage, this function supports the shape of parameter array that indicates leading dimensions of input arrays except the batch dimension.

For example, if W has the shape of $(2, 3, 4)$, x must have the shape of $(B, 2, 3, 4, S_1, ..., S_N)$ where B is the batch size and the number of trailing S's N is an arbitrary non-negative integer.

Warning: W is a trainable parameter in the original paper (https://arxiv.org/abs/1502.01852). To train W, use `chainer.links.PReLU` instead.

See also:

`chainer.links.PReLU` to manage the model parameter `W`.

chainer.functions.rrelu

`chainer.functions.rrelu(x, l=1. / 8, u=1. / 3, *, r=None, return_r=False)`

Randomized Leaky Rectified Liner Unit function.

This function is expressed as

$$f(x) = \max(x, rx),$$

where r is a random number sampled from a uniform distribution $U(l, u)$.

Note: The r corresponds to a in the original paper (https://arxiv.org/pdf/1505.00853.pdf).

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **l** (`float`) – The lower bound of the uniform distribution.
- **u** (`float`) – The upper bound of the uniform distribution.
- **r** (*N-dimensional array* or None) – The r to be used for rrelu. The shape and dtype must be the same as `x[0]` and should be on the same device. If `r` is not specified or set to `None`, an `r` will be generated randomly according to the given `l` and `u`. If `r` is specified, `l` and `u` will be ignored.
- **return_r** (`bool`) – If `True`, the r used for rrelu is returned altogether with the output variable. The returned `r` can latter be reused by passing it to `r` argument.

Returns When `return_r` is `False` (default), return the output variable. Otherwise returnes the tuple of the output variable and `r` (*N-dimensional array*). The `r` will be on the same device as the input. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable* or tuple

Example

```
>>> x = np.array([[-1, 0], [2, -3], [-2, 1]], np.float32)
>>> x
array([[-1.,  0.],
       [ 2., -3.],
       [-2.,  1.]], dtype=float32)
>>> F.rrelu(x).array
array([[-0.24850948,  0.        ],
       [ 2.        , -0.50844127],
       [-0.598535  ,  1.        ]], dtype=float32)
```

chainer.functions.relu

`chainer.functions.relu(x)`

Rectified Linear Unit function.

$$f(x) = \max(0, x).$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.array([[-1, 0], [2, -3], [-2, 1]], np.float32)
>>> np.any(x < 0)
True
>>> y = F.relu(x)
>>> np.any(y.array < 0)
False
>>> y.shape
(3, 2)
```

chainer.functions.relu6

`chainer.functions.relu6(x)`

Rectifier Unit function clipped at 6.

It computes

$$\mathrm{ReLU6}(x) = \min(\max(0, x), 6).$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_n)$-shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_n)$-shaped float array.

Return type *Variable*

See also:

chainer.functions.clipped_relu()

Example

```
>>> x = np.array([-20, -2, 0, 2, 4, 10, 100]).astype(np.float32)
>>> x
array([-20.,  -2.,   0.,   2.,   4.,  10., 100.], dtype=float32)
>>> F.relu6(x)
variable([0., 0., 0., 2., 4., 6., 6.])
```

chainer.functions.selu

`chainer.functions.selu(x, alpha=1.6732632423543772, scale=1.0507009873554805)`

Scaled Exponential Linear Unit function.

For parameters α and λ, it is expressed as

$$f(x) = \lambda \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0, \end{cases}$$

See: https://arxiv.org/abs/1706.02515

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **alpha** (*float*) – Parameter α.
- **scale** (*float*) – Parameter λ.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

chainer.functions.sigmoid

`chainer.functions.sigmoid(x)`

Element-wise sigmoid logistic function.

$$f(x) = (1 + \exp(-x))^{-1}.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

It maps the input values into the range of $[0, 1]$.

```
>>> x = np.arange(-2, 3, 2).astype(np.float32)
>>> x
array([-2.,  0.,  2.], dtype=float32)
>>> F.sigmoid(x).array
array([0.11920291, 0.5       , 0.8807971 ], dtype=float32)
```

chainer.functions.slstm

`chainer.functions.slstm(c_prev1, c_prev2, x1, x2)`

S-LSTM units as an activation function.

This function implements S-LSTM unit. It is an extension of LSTM unit applied to tree structures. The function is applied to binary trees. Each node has two child nodes. It gets four arguments, previous cell states `c_prev1` and `c_prev2`, and input arrays `x1` and `x2`.

First both input arrays `x1` and `x2` are split into eight arrays a_1, i_1, f_1, o_1, and a_2, i_2, f_2, o_2. They have the same shape along the second axis. It means that `x1` and `x2` 's second axis must have 4 times the length of `c_prev1` and `c_prev2`.

The split input arrays are corresponding to:

- a_i : sources of cell input
- i_i : sources of input gate
- f_i : sources of forget gate
- o_i : sources of output gate

It computes the updated cell state `c` and the outgoing signal `h` as:

$$c = \tanh(a_1 + a_2)\sigma(i_1 + i_2) + c_{\text{prev1}}\sigma(f_1) + c_{\text{prev2}}\sigma(f_2),$$
$$h = \tanh(c)\sigma(o_1 + o_2),$$

where σ is the elementwise sigmoid function. The function returns `c` and `h` as a tuple.

Parameters

- **c_prev1** (*Variable* or *N-dimensional array*) – Variable that holds the previous cell state of the first child node. The cell state should be a zero array or the output of the previous call of LSTM.
- **c_prev2** (*Variable* or *N-dimensional array*) – Variable that holds the previous cell state of the second child node.
- **x1** (*Variable* or *N-dimensional array*) – Variable that holds the sources of cell input, input gate, forget gate and output gate from the first child node. It must have the second dimension whose size is four times of that of the cell state.
- **x2** (*Variable* or *N-dimensional array*) – Variable that holds the input sources from the second child node.

Returns Two `Variable` objects `c` and `h`. `c` is the cell state. `h` indicates the outgoing signal.

Return type tuple

See detail in paper: Long Short-Term Memory Over Tree Structures.

Example

Assuming `c1`, `c2` is the previous cell state of children, and `h1`, `h2` is the previous outgoing signal from children. Each of `c1`, `c2`, `h1` and `h2` has `n_units` channels. Most typical preparation of `x1`, `x2` is:

```
>>> n_units = 100
>>> h1 = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> h2 = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> c1 = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> c2 = chainer.Variable(np.zeros((1, n_units), np.float32))
>>> model1 = chainer.Chain()
>>> with model1.init_scope():
...   model1.w = L.Linear(n_units, 4 * n_units)
...   model1.v = L.Linear(n_units, 4 * n_units)
>>> model2 = chainer.Chain()
```

(continues on next page)

(continued from previous page)

```
>>> with model2.init_scope():
...   model2.w = L.Linear(n_units, 4 * n_units)
...   model2.v = L.Linear(n_units, 4 * n_units)
>>> x1 = model1.w(c1) + model1.v(h1)
>>> x2 = model2.w(c2) + model2.v(h2)
>>> c, h = F.slstm(c1, c2, x1, x2)
```

It corresponds to calculate the input array `x1`, or the input sources a_1, i_1, f_1, o_1 from the previous cell state of first child node `c1`, and the previous outgoing signal from first child node `h1`. Different parameters are used for different kind of input sources.

chainer.functions.softmax

`chainer.functions.softmax(x, axis=1)`

Softmax function.

This function computes its softmax along an axis. Let $c = (c_1, c_2, \dots, c_D)$ be the slice of `x` along with the axis. For each slice c, it computes the function $f(c)$ defined as $f(c) = \frac{\exp(c)}{\sum_d \exp(c_d)}$.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A n-dimensional ($n \geq 2$) float array.
- **axis** (*int*) – The axis along which the softmax is to be computed.

Returns Output variable. A n-dimensional ($n \geq 2$) float array, which is the same shape with x.

Return type *Variable*

Example

```
>>> x = np.array([[0, 1, 2], [0, 2, 4]], np.float32)
>>> x
array([[0., 1., 2.],
       [0., 2., 4.]], dtype=float32)
>>> y = F.softmax(x, axis=1)
>>> y.array
array([[0.09003057, 0.24472848, 0.66524094],
       [0.01587624, 0.11731043, 0.86681336]], dtype=float32)
>>> F.sum(y, axis=1).array
array([1., 1.], dtype=float32)
```

chainer.functions.softplus

`chainer.functions.softplus(x, beta=1.0)`

Element-wise softplus function.

The softplus function is the smooth approximation of ReLU.

$$f(x) = \frac{1}{\beta} \log(1 + \exp(\beta x)),$$

where β is a parameter. The function becomes curved and akin to ReLU as the β is increasing.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **beta** (*float*) – Parameter β.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.arange(-2, 3, 2).astype(np.float32)
>>> x
array([-2.,  0.,  2.], dtype=float32)
>>> F.softplus(x, beta=1.0).array
array([0.126928 , 0.6931472, 2.126928 ], dtype=float32)
```

chainer.functions.swish

`chainer.functions.swish(x, beta)`

Swish activation function.

$$f(x, \beta) = x \cdot \sigma(\beta x),$$

where $\sigma(\cdot)$ is the sigmoid function. It has the following properties:

$$f(x, 0) = \frac{x}{2},$$
$$\lim_{\beta \to \infty} f(x, \beta) = \max(0, x).$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape $(s_B, s_1, s_2, ..., s_N)$, where s_B is assumed to be the *minibatch dimension*.
- **beta** (*Variable* or *N-dimensional array*) – Parameter variable β of shape $(s_1, s_2, ..., s_M)$, where M is an arbitrary integer between $0 \leq M \leq N$. The number of dimensions of `beta` will be matched with `x` by reshaping it as $(1, s_1, ..., s_M, 1, ...1)$, then `beta` and `x` are multiplied together in an element-wise manner.

Returns Output variable of the same shape as `x`.

Return type *Variable*

Warning: β is a trainable parameter in the original paper (https://arxiv.org/abs/1710.05941). To train β, use *chainer.links.Swish* instead.

See also:

chainer.links.Swish to manage the model parameter `beta`.

chainer.functions.tanh

`chainer.functions.tanh(x)`

Elementwise hyperbolic tangent function.

$$f(x) = \tanh(x).$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

Return type *Variable*

Example

```
>>> x = np.arange(-1, 4, 2).astype(np.float32)
>>> x
array([-1.,  1.,  3.], dtype=float32)
>>> F.tanh(x).array
array([-0.7615942,  0.7615942,  0.9950548], dtype=float32)
```

chainer.functions.tree_lstm

`chainer.functions.tree_lstm(*inputs)`

TreeLSTM unit as an activation function.

This function implements TreeLSTM units both for N-ary TreeLSTM and Child-Sum TreeLSTM. Let the children cell states $c_1, c_2, \dots, c_{\text{N}}$, and the incoming signal x.

First, the incoming signal x is split into (3 + N) arrays $a, i, o, f_1, f_2, ..., f_{\text{N}}$ of the same shapes along the second axis. It means that x 's second axis must have (3 + N) times of the length of each c_n.

The splitted input signals are corresponding to:

- a : sources of cell input
- i : sources of input gate
- o : sources of output gate
- f_n : sources of forget gate for n-th ary

Second, it computes outputs as:

$$\begin{aligned} c &= \tanh(a)\text{sigmoid}(i) \\ & + c_1\text{sigmoid}(f_1), \\ & + c_2\text{sigmoid}(f_2), \\ & + ..., \\ & + c_{\text{N}}\text{sigmoid}(f_{\text{N}}), \\ h &= \tanh(c)\text{sigmoid}(o). \end{aligned}$$

These are returned as a tuple of (N + 1) variables.

Parameters **inputs** (list of *Variable*) – Variable arguments which include all cell vectors from child-nodes, and an input vector. Each of the cell vectors and the input vector is *Variable* or *N-dimensional array*. The input vector must have the second dimension whose size is (N + 3) times of that of each cell, where N denotes the total number of cells.

Returns Two *Variable* objects `c` and `h`. `c` is the updated cell state. `h` indicates the outgoing signal.

Return type tuple

See the papers for details: Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks and A Fast Unified Model for Parsing and Sentence Understanding.

Tai et al.'s N-Ary TreeLSTM is little extended in Bowman et al., and this link is based on the variant by Bowman et al. Specifically, eq. 10 in Tai et al. only has one W matrix to be applied to x, consistently for all children. On the other hand, Bowman et al.'s model has multiple matrices, each of which affects the forget gate for each child's cell individually.

Example

Assuming `y` is the current input signal, `c` is the previous cell state, and `h` is the previous output signal from an *tree_lstm()* function. Each of `y`, `c` and `h` has `n_units` channels. Using 2-ary (binary) TreeLSTM, most typical preparation of `x` is:

```
>>> model = chainer.Chain()
>>> with model.init_scope():
...   model.w = L.Linear(10, 5 * 10)
...   model.v1 = L.Linear(10, 5 * 10)
...   model.v2 = L.Linear(10, 5 * 10)
>>> y = np.random.uniform(-1, 1, (4, 10)).astype(np.float32)
>>> h1 = np.random.uniform(-1, 1, (4, 10)).astype(np.float32)
>>> h2 = np.random.uniform(-1, 1, (4, 10)).astype(np.float32)
>>> c1 = np.random.uniform(-1, 1, (4, 10)).astype(np.float32)
>>> c2 = np.random.uniform(-1, 1, (4, 10)).astype(np.float32)
>>> x = model.w(y) + model.v1(h1) + model.v2(h2)
>>> c, h = F.tree_lstm(c1, c2, x)
```

It corresponds to calculate the input sources a, i, o, f_1, f_2 from the current input `y` and the children's outputs `h1` and `h2`. Different parameters are used for different kind of input sources.

4.2.3 Array manipulations

chainer.functions.as_strided	Create a new view of array with the given shape, strides, and offset.
chainer.functions.broadcast	Broadcast given variables.
chainer.functions.broadcast_to	Broadcast a given variable to a given shape.
chainer.functions.cast	Cast an input variable to a given type.
chainer.functions.concat	Concatenates given variables along an axis.
chainer.functions.copy	Copies the input variable onto the specified device.
chainer.functions.depth2space	Computes the depth2space transformation for subpixel calculations.
chainer.functions.diagonal	Take diagonal
chainer.functions.dstack	Concatenate variables along third axis (depth wise).

Continued on next page

Table 4 – continued from previous page

chainer.functions.expand_dims	Expands dimensions of an input variable without copy.
chainer.functions.flatten	Flatten a given array into one dimension.
chainer.functions.flip	Flips an input variable in reverse order along the given axis.
chainer.functions.fliplr	Flip array in the left/right direction.
chainer.functions.flipud	Flip array in the up/down direction.
chainer.functions.get_item	Extract elements from array with specified shape, axes and offsets.
chainer.functions.hstack	Concatenate variables horizontally (column wise).
chainer.functions.im2col	Extract patches from an image based on the filter.
chainer.functions.moveaxis	Move the source axes to the destination.
chainer.functions.pad	Pad an input variable.
chainer.functions.pad_sequence	Pad given arrays to make a matrix.
chainer.functions.permutate	Permutates a given variable along an axis.
chainer.functions.repeat	Construct an array by repeating a given array.
chainer.functions.reshape	Reshapes an input variable without copy.
chainer.functions.resize_images	Resize images to the given shape.
chainer.functions.rollaxis	Roll the axis backwards to the given position.
chainer.functions.scatter_add	Adds given values to specified elements of an array.
chainer.functions.select_item	Select elements stored in given indices.
chainer.functions.separate	Separates an array along a given axis.
chainer.functions.space2depth	Computes the space2depth transformation for subpixel calculations.
chainer.functions.spatial_transformer_grid	2D Spatial Transformer grid.
chainer.functions.spatial_transformer_sampler	2D Spatial Transformer sampler.
chainer.functions.split_axis	Splits given variables along an axis.
chainer.functions.squeeze	Remove dimensions of size one from the shape of a ndarray.
chainer.functions.stack	Concatenate variables along a new axis.
chainer.functions.swapaxes	Swap two axes of a variable.
chainer.functions.tile	Construct an array by tiling a given array.
chainer.functions.transpose	Permute the dimensions of an input variable without copy.
chainer.functions.transpose_sequence	Transpose a list of Variables.
chainer.functions.vstack	Concatenate variables vertically (row wise).
chainer.functions.where	Choose elements depending on condition.

chainer.functions.as_strided

chainer.functions.**as_strided**(*x*, *shape*, *strides*, *storage_offset=None*)

Create a new view of array with the given shape, strides, and offset.

Parameters

- **x** (tuple of *Variable* or numpy.ndarray or cupy.ndarray) – The array pointing a memory buffer. Its view is totally ignored.
- **shape** (*tuple of int*) – The shape of output.
- **strides** (*tuple of int*) – The strides of output, given in the unit of steps.
- **storage_offset** (*int*) – The offset between the head of allocated memory and the

pointer of first element, given in the unit of steps.

Returns The strided variable.

Return type *Variable*

Warning: Users should be aware that this function potentially causes unintended side effects. See numpy.lib.stride_tricks.as_strided for the detail.

Note: The backward algorithm is borrowed from *torch.Tensor.as_strided*. Therefore, the returned gradient of `backward` is *layout-agnostic* when `x` contains memory overlap. See notes in pytorch's source code (as_strided Backward and layout-aware/agnostic autograd) too.

Note: In this function `strides` and `storage_offset` are given in the unit of steps instead of bytes. This specification differs from `numpy.lib.stride_tricks.as_strided()`.

Example

```
>>> from chainer import functions as F, Variable
>>> x = Variable(np.arange(4, dtype=np.float32))
>>> x
variable([0., 1., 2., 3.])
>>> y = F.as_strided(x, (3, 2), (1, 1), 0)
>>> y
variable([[0., 1.],
          [1., 2.],
          [2., 3.]])
>>> y.grad = np.ones((3, 2), dtype=np.float32)
>>> y.backward()
>>> x.grad
array([1., 2., 2., 1.], dtype=float32)
```

chainer.functions.broadcast

`chainer.functions.broadcast(*args)`

Broadcast given variables.

Parameters **args** (`Variable` or *N-dimensional array*) – Input variables to be broadcasted. Each dimension of the shapes of the input variables must have the same size.

Returns `Variable` or tuple of `Variable` objects which are broadcasted from the given arguments.

Return type *Variable*

Example

```
>>> x = np.random.uniform(0, 1, (3, 2)).astype(np.float32)
>>> y = F.broadcast(x)
```

(continues on next page)

(continued from previous page)

```
>>> np.all(x == y.array)
True
>>> z = np.random.uniform(0, 1, (3, 2)).astype(np.float32)
>>> y, w = F.broadcast(x, z)
>>> np.all(x == y.array) & np.all(z == w.array)
True
```

chainer.functions.broadcast_to

`chainer.functions.broadcast_to(x, shape)`

Broadcast a given variable to a given shape.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable to be broadcasted. A $(s_1, s_2, ..., s_N)$-shaped float array.
- **shape** (`tuple`) – Tuple of `int` of the shape of the output variable.

Returns Output variable broadcasted to the given shape.

Return type *Variable*

Example

```
>>> x = np.arange(0, 3)
>>> x
array([0, 1, 2])
>>> y = F.broadcast_to(x, (3, 3))
>>> y.array
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

chainer.functions.cast

`chainer.functions.cast(x, typ)`

Cast an input variable to a given type.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable to be casted. A $(s_1, s_2, ..., s_N)$-shaped array.
- **typ** (`str` of dtype or `numpy.dtype`) – Typecode or data type to cast.

Returns Variable holding a casted array.

Return type *Variable*

Example

```
>>> x = np.arange(0, 3, dtype=np.float64)
>>> x.dtype
dtype('float64')
>>> y = F.cast(x, np.float32)
>>> y.dtype
dtype('float32')
>>> y = F.cast(x, 'float16')
>>> y.dtype
dtype('float16')
```

chainer.functions.concat

`chainer.functions.concat(xs, axis=1)`

Concatenates given variables along an axis.

Parameters

- **xs** (tuple of `Variable` or *N-dimensional array*) – Input variables to be concatenated. The variables must have the same shape, except in the dimension corresponding to axis.
- **axis** (*int*) – The axis along which the arrays will be joined. Default is 1.

Returns The concatenated variable.

Return type *Variable*

Example

```
>>> x = np.arange(0, 12).reshape(3, 4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> y = np.arange(0, 3).reshape(3, 1)
>>> y
array([[0],
       [1],
       [2]])
>>> z = F.concat((x, y), axis=1)
>>> z.array
array([[ 0,  1,  2,  3,  0],
       [ 4,  5,  6,  7,  1],
       [ 8,  9, 10, 11,  2]])
```

chainer.functions.copy

`chainer.functions.copy(x, dst)`

Copies the input variable onto the specified device.

If the input `x` already resides on the device specified by `dst`, no copy will actually take place and the returned variable will hold a view of the input. In other cases, the input will be copied to `dst`. When `dst == -1`, the array is copied to the host memory. This function supports copies from host to host, from host to device, from device to device and from device to host.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Variable to be copied.
- **dst** – Target device specifier.

Returns Output variable.

Return type *Variable*

Example

```
>>> import chainer.backends.cuda as cuda
>>> x_arr = np.random.uniform(-1, 1, (5, 10))
>>> x = chainer.Variable(x_arr)
>>> x.device
<CpuDevice (numpy)>
>>> y = F.copy(x, '@cupy:0') # from CPU (NumPy) to GPU 0 (CuPy)
>>> y.device
<GpuDevice (cupy):0>
```

Note: Copies between non-ChainerX devices and ChainerX devices are not supported.

chainer.functions.depth2space

`chainer.functions.depth2space(X, r)`

Computes the depth2space transformation for subpixel calculations.

Parameters

- **X** (*Variable* or *N-dimensional array*) – Variable holding a 4d array of shape `(batch, channel * r * r, dim1, dim2)`.
- **r** (*int*) – the upscaling factor.

Returns A variable holding the upscaled array from interspersed depth layers. The shape is `(batch, channel, dim1 * r, dim2 * r)`.

Return type *Variable*

Note: This can be used to compute super-resolution transformations. See https://arxiv.org/abs/1609.05158 for details.

See also:

space2depth()

Example

```
>>> X = np.arange(24).reshape(1, 4, 2, 3).astype(np.float32)
>>> X.shape
(1, 4, 2, 3)
>>> X
array([[[[ 0.,  1.,  2.],
```

(continues on next page)

(continued from previous page)

```
         [ 3.,  4.,  5.]],

        [[ 6.,  7.,  8.],
         [ 9., 10., 11.]],

        [[12., 13., 14.],
         [15., 16., 17.]],

        [[18., 19., 20.],
         [21., 22., 23.]]]], dtype=float32)
>>> y = F.depth2space(X, 2)
>>> y.shape
(1, 1, 4, 6)
>>> y.array
array([[[[ 0.,  6.,  1.,  7.,  2.,  8.],
         [12., 18., 13., 19., 14., 20.],
         [ 3.,  9.,  4., 10.,  5., 11.],
         [15., 21., 16., 22., 17., 23.]]]], dtype=float32)
```

chainer.functions.diagonal

`chainer.functions.diagonal(x, offset=0, axis1=0, axis2=1)`

Take diagonal

Axes other than `axis1` and `axis2` are regarded as batch dimensions.

Parameters

- **x** (*Variable* or *N-dimensional array*) – A variable to be sliced.
- **offset** (*int*) – Offset from the principal diagonal. An upper diagonal matrix can have nonzero diagonals with nonnegative offsets.
- **axis1** (*int*) – First axis (that has row indices) of matrix
- **axis2** (*int*) – Second axis (that has column indices) of matrix

Returns (Batched) diagonal vectors

Return type *Variable*

Example

```
>>> x = chainer.Variable(np.arange(9).reshape(3, 3).astype(np.float32))
>>> x
variable([[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]])
>>> chainer.functions.diagonal(x, offset=1)
variable([1., 5.])
```

chainer.functions.dstack

`chainer.functions.dstack(xs)`

Concatenate variables along third axis (depth wise).

Parameters **xs** (list of `Variable` or *N-dimensional array*) – Input variables to be concatenated. The variables must have the same `ndim`. When the variables have the third axis (i.e. $ndim \geq 3$), the variables must have the same shape along all but the third axis. When the variables do not have the third axis(i.e. $ndim < 3$), the variables must have the same shape.

Returns Output variable. When the input variables have the third axis (i.e. $ndim \geq 3$), the shapes of inputs and output are the same along all but the third axis. The length of third axis is the sum of the lengths of inputs' third axis. When the shape of variables are `(N1, N2)` (i.e. $ndim = 2$), the shape of output is `(N1, N2, 2)`. When the shape of variables are `(N1,)` (i.e. $ndim = 1$), the shape of output is `(1, N1, 2)`. When the shape of variables are `()` (i.e. $ndim = 0$), the shape of output is `(1, 1, 2)`.

Return type *Variable*

Example

```
>>> x1 = np.array((1, 2, 3))
>>> x1.shape
(3,)
>>> x2 = np.array((2, 3, 4))
>>> x2.shape
(3,)
>>> y = F.dstack((x1, x2))
>>> y.shape
(1, 3, 2)
>>> y.array
array([[[1, 2],
        [2, 3],
        [3, 4]]])
```

```
>>> x1 = np.arange(0, 6).reshape(3, 2)
>>> x1.shape
(3, 2)
>>> x1
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> x2 = np.arange(6, 12).reshape(3, 2)
>>> x2.shape
(3, 2)
>>> x2
array([[ 6,  7],
       [ 8,  9],
       [10, 11]])
>>> y = F.dstack([x1, x2])
>>> y.shape
(3, 2, 2)
>>> y.array
array([[[ 0,  6],
        [ 1,  7]],

       [[ 2,  8],
```

(continues on next page)

(continued from previous page)

```
        [ 3,  9]],

       [[ 4, 10],
        [ 5, 11]]])
```

```
>>> x1 = np.arange(0, 12).reshape(3, 2, 2)
>>> x2 = np.arange(12, 18).reshape(3, 2, 1)
>>> y = F.dstack([x1, x2])
>>> y.shape
(3, 2, 3)
>>> y.array
array([[[ 0,  1, 12],
        [ 2,  3, 13]],

       [[ 4,  5, 14],
        [ 6,  7, 15]],

       [[ 8,  9, 16],
        [10, 11, 17]]])
```

chainer.functions.expand_dims

`chainer.functions.expand_dims(x, axis)`

Expands dimensions of an input variable without copy.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **axis** (*int*) – Position where new axis is to be inserted. The `axis` parameter is acceptable when $-ndim - 1 \leq axis \leq ndim$. (`ndim` is the dimension of input variables). When $axis < 0$, the result is the same with $ndim + 1 - |axis|$.

Returns Variable that holds an expanded input. The `ndim` of output is one greater than that of `x`.

Return type *Variable*

Example

```
>>> x = np.array([1, 2, 3])
>>> x.shape
(3,)
>>> y = F.expand_dims(x, axis=0)
>>> y.shape
(1, 3)
>>> y.array
array([[1, 2, 3]])
>>> y = F.expand_dims(x, axis=1)
>>> y.shape
(3, 1)
>>> y.array
array([[1],
       [2],
       [3]])
```

(continues on next page)

(continued from previous page)

```
>>> y = F.expand_dims(x, axis=-2)
>>> y.shape
(1, 3)
>>> y.array
array([[1, 2, 3]])
```

chainer.functions.flatten

chainer.functions.**flatten**(*x*)

Flatten a given array into one dimension.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable flatten to one dimension.

Return type *Variable*

Note: When you input a scalar array (i.e. the shape is `()`), you can also get the one dimension array whose shape is `(1,)`.

Example

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x.shape
(2, 2)
>>> y = F.flatten(x)
>>> y.shape
(4,)
>>> y.array
array([1, 2, 3, 4])
```

```
>>> x = np.arange(8).reshape(2, 2, 2)
>>> x.shape
(2, 2, 2)
>>> y = F.flatten(x)
>>> y.shape
(8,)
>>> y.array
array([0, 1, 2, 3, 4, 5, 6, 7])
```

chainer.functions.flip

chainer.functions.**flip**(*x*, *axis*)

Flips an input variable in reverse order along the given axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **axis** (*int*) – Axis along which the input variable is reversed.

Returns Output variable.

Return type *Variable*

chainer.functions.fliplr

`chainer.functions.`**`fliplr`**`(`*a*`)`

Flip array in the left/right direction.

Parameters **a** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.flipud

`chainer.functions.`**`flipud`**`(`*a*`)`

Flip array in the up/down direction.

Parameters **a** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.get_item

`chainer.functions.`**`get_item`**`(`*x*, *slices*`)`

Extract elements from array with specified shape, axes and offsets.

Parameters

- **x** (*Variable* or *N-dimensional array*) – A variable to be sliced.
- **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – An object to specify the selection of elements.

Returns A *Variable* object which contains sliced array of `x`.

Note: It only supports types that are supported by CUDA's atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

Note: It does not support `slices` that contains multiple boolean arrays.

Note: See NumPy documentation for details of indexing.

Example

```
>>> x = np.arange(12).reshape((2, 2, 3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
>>> F.get_item(x, 0)
variable([[0, 1, 2],
          [3, 4, 5]])
>>> F.get_item(x, (0, 0, slice(0, 2, 1)))  # equals x[0, 0, 0:2:1]
variable([0, 1])
>>> F.get_item(x, (Ellipsis, 2))  # equals x[..., 2]
variable([[ 2,  5],
          [ 8, 11]])
>>> F.get_item(x, (1, np.newaxis, 1, 0))  # equals x[1, None, 1, 0]
variable([9])
```

chainer.functions.hstack

chainer.functions.**hstack**(*xs*)

Concatenate variables horizontally (column wise).

Parameters **xs** (list of `Variable` or *N-dimensional array*) – Input variables to be concatenated. The variables must have the same `ndim`. When the variables have the second axis (i.e. $ndim \geq 2$), the variables must have the same shape along all but the second axis. When the variables do not have the second axis(i.e. $ndim < 2$), the variables need not to have the same shape.

Returns Output variable. When the input variables have the second axis (i.e. $ndim \geq 2$), the shapes of inputs and output are the same along all but the second axis. The length of second axis is the sum of the lengths of inputs' second axis. When the variables do not have the second axis (i.e. $ndim < 2$), the shape of output is `(N, )` (`N` is the sum of the input variables' size).

Return type *Variable*

Example

```
>>> x1 = np.array((1, 2, 3))
>>> x1.shape
(3,)
>>> x2 = np.array((2, 3, 4))
>>> x2.shape
(3,)
>>> y = F.hstack((x1, x2))
>>> y.shape
(6,)
>>> y.array
array([1, 2, 3, 2, 3, 4])
>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

(continues on next page)

(continued from previous page)

```
       [ 8,  9, 10, 11]])
>>> x2 = np.arange(12, 18).reshape(3, 2)
>>> x2.shape
(3, 2)
>>> x2
array([[12, 13],
       [14, 15],
       [16, 17]])
>>> y = F.hstack([x1, x2])
>>> y.shape
(3, 6)
>>> y.array
array([[ 0,  1,  2,  3, 12, 13],
       [ 4,  5,  6,  7, 14, 15],
       [ 8,  9, 10, 11, 16, 17]])
```

chainer.functions.im2col

`chainer.functions.im2col`(*x*, *ksize*, *stride=1*, *pad=0*, *cover_all=False*, *dilate=1*)

Extract patches from an image based on the filter.

This function rearranges patches of an image and puts them in the channel dimension of the output.

Patches are extracted at positions shifted by multiples of `stride` from the first position `-pad` for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Notation: here is a notation.

- n is the batch size.
- c is the number of the input channels.
- h and w are the height and width of the input image, respectively.
- k_H and k_W are the height and width of the filters, respectively.
- s_Y and s_X are the strides of the filter.
- p_H and p_W are the spatial padding sizes.
- d_Y and d_X are the dilation factors of filter application.

The output size (h_O, w_O) is determined by the following equations when `cover_all = False`:

$$
\begin{aligned}
h_O &= (h + 2p_H - k_H - (k_H - 1) * (d_Y - 1))/s_Y + 1,\\
w_O &= (w + 2p_W - k_W - (k_W - 1) * (d_X - 1))/s_X + 1.
\end{aligned}
$$

When `cover_all = True`, the output size is determined by the following equations:

$$
\begin{aligned}
h_O &= (h + 2p_H - k_H - (k_H - 1) * (d_Y - 1) + s_Y - 1)/s_Y + 1,\\
w_O &= (w + 2p_W - k_W - (k_W - 1) * (d_X - 1) + s_X - 1)/s_X + 1.
\end{aligned}
$$

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape (n, c, h, w).
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.

- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **cover_all** (*bool*) – If `True`, all spatial locations are rearranged into some output pixels. It may make the output size larger.
- **dilate** (*int or pair of ints*) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.

Returns Output variable whose shape is $(n, c \cdot k_H \cdot k_W, h_O, w_O)$

Return type *Variable*

chainer.functions.moveaxis

`chainer.functions.moveaxis(x, source, destination)`

Move the source axes to the destination.

This function transpose the input `x` by moving the axes `source` to the axes `destination`. Other axes remain in their original order.

See also `chainer.functions.transpose()`, `chainer.functions.swapaxes()`.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **source** (*int or tuple of int*) – Original positions of the axes to move. These must be unique.
- **destination** (*int or tuple of int*) – Destination positions for each of the original axes. These must also be unique.

Returns Variable whose axis is moved.

Return type *Variable*

Example

```
>>> x = np.zeros((2, 3, 4, 5), np.float32)
>>> chainer.functions.moveaxis(x, 0, -1).shape
(3, 4, 5, 2)
>>> chainer.functions.moveaxis(x, (0, 3), (2, 0)).shape
(5, 3, 2, 4)
```

chainer.functions.pad

`chainer.functions.pad(x, pad_width, mode, **keywords)`

Pad an input variable.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input data.
- **pad_width** (*int or array-like*) – Number of values padded to the edges of each axis.

- **mode** (*str*) – Specifies how the function fills the periphery of the array. The mode is passed to `numpy.pad()` or `cupy.pad()`. If it is `'constant'`, the input is padded by a constant value specified by `constant_values`.
- **constant_values** (*int or array-like*) – Constant values to fill the periphery in the `'constant'` mode.

Returns Output variable.

Return type *Variable*

chainer.functions.pad_sequence

`chainer.functions.pad_sequence(xs, length=None, padding=0)`

Pad given arrays to make a matrix.

Parameters

- **xs** (list of ~chainer.Variable or *N-dimensional array*) – Variables you want to concatenate.
- **length** (*None or int*) – Size of the first dimension of a padded array. If it is `None`, the longest size of the first dimension of `xs` is used.
- **padding** (*int or float*) – Value to fill.

Returns A padded matrix. Its shape is `(n, length, ...)`, where `n == len(xs)`.

Return type *Variable*

chainer.functions.permutate

`chainer.functions.permutate(x, indices, axis=0, inv=False)`

Permutates a given variable along an axis.

This function permutate `x` with given `indices`. That means `y[i] = x[indices[i]]` for all `i`. Note that this result is same as `y = x.take(indices)`. `indices` must be a permutation of `[0, 1, ..., len(x) - 1]`.

When `inv` is `True`, `indices` is treated as its inverse. That means `y[indices[i]] = x[i]`.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Variable to permutate. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **indices** (`Variable` or *N-dimensional array*) – Indices to extract from the variable. A one-dimensional int array.
- **axis** (*int*) – Axis that the input array is permutate along.
- **inv** (*bool*) – If `True`, `indices` is treated as its inverse.

Returns Output variable.

Return type *Variable*

Example

```
>>> x = np.arange(6).reshape((3, 2)).astype(np.float32)
>>> x
array([[0., 1.],
       [2., 3.],
       [4., 5.]], dtype=float32)
>>> indices = np.array([2, 0, 1], np.int32)
>>> y = F.permutate(x, indices)
>>> y.array
array([[4., 5.],
       [0., 1.],
       [2., 3.]], dtype=float32)
>>> y = F.permutate(x, indices, inv=True)
>>> y.array
array([[2., 3.],
       [4., 5.],
       [0., 1.]], dtype=float32)
>>> indices = np.array([1, 0], np.int32)
>>> y = F.permutate(x, indices, axis=1)
>>> y.array
array([[1., 0.],
       [3., 2.],
       [5., 4.]], dtype=float32)
```

chainer.functions.repeat

`chainer.functions.repeat(x, repeats, axis=None)`

Construct an array by repeating a given array.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **repeats** (`int` or `tuple` of `int` s) – The number of times which each element of `x` is repeated.
- **axis** (`int`) – The axis along which to repeat values.

Returns The repeated output Variable.

Return type *Variable*

Example

```
>>> x = np.array([0, 1, 2])
>>> x.shape
(3,)
>>> y = F.repeat(x, 2)
>>> y.shape
(6,)
>>> y.array
array([0, 0, 1, 1, 2, 2])
>>> x = np.array([[1,2], [3,4]])
>>> x.shape
(2, 2)
>>> y = F.repeat(x, 3, axis=1)
>>> y.shape
```

(continues on next page)

(continued from previous page)

```
(2, 6)
>>> y.array
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
>>> y = F.repeat(x, (1, 2), axis=0)
>>> y.shape
(3, 2)
>>> y.array
array([[1, 2],
       [3, 4],
       [3, 4]])
```

chainer.functions.reshape

`chainer.functions.reshape(x, shape)`

Reshapes an input variable without copy.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **shape** (`tuple` of `int` s) – Expected shape of the output array. The number of elements which the array of `shape` contains must be equal to that of input array. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

Returns Variable that holds a reshaped version of the input variable.

Return type *Variable*

See also:

`numpy.reshape()`, `cupy.reshape()`

Example

```
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
>>> y = F.reshape(x, (8,))
>>> y.shape
(8,)
>>> y.array
array([1, 2, 3, 4, 5, 6, 7, 8])
>>> y = F.reshape(x, (4, -1))  # the shape of output is inferred
>>> y.shape
(4, 2)
>>> y.array
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> y = F.reshape(x, (4, 3))  # the shape of input and output are not consistent
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType:
Invalid operation is performed in: Reshape (Forward)
```

(continues on next page)

(continued from previous page)

```
Expect: prod(in_types[0].shape) == prod((4, 3))
Actual: 8 != 12
```

chainer.functions.resize_images

`chainer.functions.resize_images(x, output_shape, mode='bilinear', align_corners=True)`

Resize images to the given shape.

This function resizes 2D data to `output_shape`. Currently, only bilinear interpolation is supported as the sampling method.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I is the number of the input channels.
- h and w are the height and width of the input image, respectively.
- h_O and w_O are the height and width of the output image.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **output_shape** (*tuple*) – This is a tuple of length 2 whose values are `(h_O, w_O)`. Note that the order of height and width is opposite of the one in OpenCV.
- **mode** (*{'bilinear', 'nearest'}*) – Defines the sampling rule.
- **align_corners** (*bool*) – When this value is `True`, the corners of the input are mapped to the corners of the output. When `False`, the behavior is the same as OpenCV.

Returns Resized image whose shape is (n, c_I, h_O, w_O).

Return type *Variable*

chainer.functions.rollaxis

`chainer.functions.rollaxis(x, axis, start=0)`

Roll the axis backwards to the given position.

This function continues to be supported for backward compatibility, but you should prefer `chainer.functions.moveaxis(x, source, destination)`. See `chainer.functions.moveaxis()`.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **axis** (*int*) – The axis to roll backwards.
- **start** (*int*) – The place to which the axis is moved.

Returns Variable whose axis is rolled.

Return type *Variable*

chainer.functions.scatter_add

`chainer.functions.scatter_add(a, slices, b)`

Adds given values to specified elements of an array.

This function adds `b` to the specified elements of the copy of `a`, and returns the copy. The value of the original `a` is not changed.

Parameters

- **a** (*Variable* or *N-dimensional array*) – A variable.
- **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – It is an integer, a slice, an ellipsis, a numpy.newaxis, an integer array-like, a boolean array-like or tuple of them.
- **b** (*Variable* or *N-dimensional array*) – A variable that is scatter added to `a`. Its shape has to equal `a[slices]` because broadcasting of variables is not supported.

Returns A *Variable* object which is the result of scatter addition.

Note: It only supports types that are supported by CUDA's atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

Note: It does not support `slices` that contains multiple boolean arrays.

See also:

`numpy.add.at()` and `cupyx.scatter_add()`.

chainer.functions.select_item

`chainer.functions.select_item(x, t)`

Select elements stored in given indices.

This function returns `t.choose(x.T)`, that means `y[i] == x[i, t[i]]` for all `i`.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Variable storing arrays. A two-dimensional float array.
- **t** (*Variable* or *N-dimensional array*) – Variable storing index numbers. A one-dimensional int array. Length of the `t` should be equal to `x.shape[0]`.

Returns Variable that holds `t`-th element of `x`.

Return type *Variable*

Example

```
>>> x = np.array([[0, 1, 2], [3, 4, 5]], np.float32)
>>> t = np.array([0, 2], np.int32)
>>> y = F.select_item(x, t)
>>> y.shape
```

(continues on next page)

(continued from previous page)

```
(2,)
>>> y.array
array([0., 5.], dtype=float32)
```

chainer.functions.separate

`chainer.functions.separate(x, axis=0)`

Separates an array along a given axis.

This function separates an array along a given axis. For example, shape of an array is `(2, 3, 4)`. When it separates the array with `axis=1`, it returns three `(2, 4)` arrays.

This function is an inverse of `chainer.functions.stack()`.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Variable to be separated. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **axis** (`int`) – Axis along which variables are separated.

Returns Output variables.

Return type tuple of chainer.Variable

See also:

`chainer.functions.stack()`

Example

```
>>> x = np.arange(6).reshape((2, 3)).astype(np.float32)
>>> x
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
>>> x.shape
(2, 3)
>>> y = F.separate(x) # split along axis=0
>>> isinstance(y, tuple)
True
>>> len(y)
2
>>> y[0].shape
(3,)
>>> y[0].array
array([0., 1., 2.], dtype=float32)
>>> y = F.separate(x, axis=1)
>>> len(y)
3
>>> y[0].shape
(2,)
>>> y[0].array
array([0., 3.], dtype=float32)
```

chainer.functions.space2depth

`chainer.functions.space2depth(X, r)`

Computes the space2depth transformation for subpixel calculations.

Parameters

- **X** (`Variable` or *N-dimensional array*) – Variable holding a 4d array of shape `(batch, channel, dim1 * r, dim2 * r)`.
- **r** (`int`) – the downscaling factor.

Returns A variable holding the downscaled layer array from subpixel array sampling. The shape is `(batch, channel * r * r, dim1, dim2)`.

Return type *Variable*

Note: This can be used to compute inverse super-resolution transformations. See https://arxiv.org/abs/1609.05158 for details.

See also:

`depth2space()`

Example

```
>>> X = np.arange(24).reshape(1, 1, 4, 6).astype(np.float32)
>>> X.shape
(1, 1, 4, 6)
>>> X
array([[[[ 0.,  1.,  2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9., 10., 11.],
         [12., 13., 14., 15., 16., 17.],
         [18., 19., 20., 21., 22., 23.]]]], dtype=float32)
>>> y = F.space2depth(X, 2)
>>> y.shape
(1, 4, 2, 3)
>>> y.array
array([[[[ 0.,  2.,  4.],
         [12., 14., 16.]],

        [[ 1.,  3.,  5.],
         [13., 15., 17.]],

        [[ 6.,  8., 10.],
         [18., 20., 22.]],

        [[ 7.,  9., 11.],
         [19., 21., 23.]]]], dtype=float32)
```

chainer.functions.spatial_transformer_grid

`chainer.functions.spatial_transformer_grid(theta, output_shape, **kwargs)`

2D Spatial Transformer grid.

This function generates coordinates of the points sampled from an image to perform warping described in Spatial Transformer Networks.

Given a coordinate in the warped image (x_i^t, y_i^t), the point sampled from the source image (x_i^s, y_i^s) are calculated by the following equation.

Note: cuDNN supports SpatialTransformerGrid from version 5.0.0.

$$\begin{pmatrix} x_i^s \\ y_i^s \end{pmatrix} = \begin{pmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{pmatrix} \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix}$$

Notation: here is a notation for dimensionalities.

- n is the batch size.
- h_O and w_O are the height and the width of the output image.

Parameters

- **theta** (*Variable* or *N-dimensional array*) – An array of shape $(n, 2, 3)$. This is a batch of 2×3 matrix used for the warping described above.
- **output_shape** (*tuple*) – A tuple of 2 elements: h_O, w_O.

Returns A variable of shape $(n, 2, h_O, w_O)$. In the 2nd dimension, the first element is the coordinate along the x axis, and the second element is the coordinate along the y axis. All the coordinates in the image are scaled to fit range $[-1, 1]$. This means that the coordinate $(-1, -1)$ corresponds to the upper-left corner of the input image.

Return type *Variable*

chainer.functions.spatial_transformer_sampler

`chainer.functions.spatial_transformer_sampler(x, grid, **kwargs)`

2D Spatial Transformer sampler.

This is a differentiable image sampler. With a set of sampling points `grid` and an input feature map `x`, this produces a sampled output feature map.

This function currently only supports bilinear interpolation as a sampling kernel.

When coordinates in `grid` is outside range $[-1, 1]$, values are sampled from a zero padded input image.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I is the number of the input channels.
- h and w are the height and width of the input image, respectively.
- h_O and w_O are the height and width of the output image.

See detail in the following paper: Spatial Transformer Networks.

Note: cuDNN supports SpatialTransformerSampler from version 5.0.0.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **grid** (Variable) – Coordinate variable of shape $(n, 2, h_O, w_O)$. Each coordinate defines the spatial location in the input where a sampling kernel is applied to get the value at a particular pixel in the output. `grid[idx, :, i, j]` corresponds to the coordinate that is used to sample the values for an output pixel at location (i, j).

  In the second dimension, the first coordinate corresponds to the location along the horizontal axis, and the second coordinate corresponds to the location along the vertical axis.

  The coordinate $(-1, -1)$ corresponds to the upper-left corner of the input image.

Returns Output feature map of shape (n, c_I, h_O, w_O).

Return type *Variable*

chainer.functions.split_axis

`chainer.functions.split_axis(x, indices_or_sections, axis, force_tuple=True)`

Splits given variables along an axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – A variable to be split.
- **indices_or_sections** (*int or 1-D array*) – If this argument is an integer, N, the array will be divided into N equal arrays along axis. If it is a 1-D array of sorted integers, it indicates the positions where the array is split.
- **axis** (*int*) – Axis that the input array is split along.
- **force_tuple** (*bool*) – If `True` (the default) this method returns a tuple even when the number of outputs is one. Otherwise, if `False` a Variable will be returned when the number of outputs is one.

Returns Tuple of *Variable* objects if the number of outputs is more than 1 or *Variable* otherwise. When `force_tuple` is `True`, returned value is always a tuple regardless of the number of outputs.

Return type tuple or *Variable*

chainer.functions.squeeze

`chainer.functions.squeeze(x, axis=None)`

Remove dimensions of size one from the shape of a ndarray.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **axis** (*None or int or tuple of ints*) – A subset of the single-dimensional entries in the shape to remove. If `None` is supplied, all of them are removed. The dimension index starts at zero. If an axis with dimension greater than one is selected, an error is raised.

Returns Variable whose dimensions of size 1 are removed.

Return type *Variable*

Example

```
>>> x = np.array([[[[0, 1, 2]]], [[[3, 4, 5]]]], np.float32)
>>> x.shape
(2, 1, 1, 3)
>>> y = F.squeeze(x)
>>> y.shape
(2, 3)
>>> y.array
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
>>> y = F.squeeze(x, axis=1)
>>> y.shape
(2, 1, 3)
>>> y.array
array([[[0., 1., 2.]],

       [[3., 4., 5.]]], dtype=float32)
>>> y = F.squeeze(x, axis=(1, 2))
>>> y.shape
(2, 3)
>>> y.array
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
```

chainer.functions.stack

`chainer.functions.stack(xs, axis=0)`

Concatenate variables along a new axis.

Parameters

- **xs** (list of `Variable` or *N-dimensional array*) – Input variables to be concatenated. The variables must have the same shape.
- **axis** (*int*) – The axis along which the arrays will be stacked. The `axis` parameter is acceptable when $-ndim - 1 \leq axis \leq ndim$. (`ndim` is the dimension of input variables). When $axis < 0$, the result is the same with $ndim + 1 - |axis|$.

Returns Output variable. Let `x_1, x_2, ..., x_n` and `y` be the input variables and the output variable, `y[:, ..., 0, ..., :]` is `x_1`, `y[:, ..., 1, ..., :]` is `x_2` and `y[:, ..., n-1, ..., :]` is `x_n` (The indexed axis indicates the `axis`).

Return type *Variable*

Example

```
>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> x2 = np.arange(12, 24).reshape(3, 4)
```

(continues on next page)

(continued from previous page)

```
>>> x2.shape
(3, 4)
>>> x2
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> y = F.stack([x1, x2], axis=0)
>>> y.shape
(2, 3, 4)
>>> y.array
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y = F.stack([x1, x2], axis=1)
>>> y.shape
(3, 2, 4)
>>> y.array
array([[[ 0,  1,  2,  3],
        [12, 13, 14, 15]],

       [[ 4,  5,  6,  7],
        [16, 17, 18, 19]],

       [[ 8,  9, 10, 11],
        [20, 21, 22, 23]]])
>>> y = F.stack([x1, x2], axis=2)
>>> y.shape
(3, 4, 2)
>>> y.array
array([[[ 0, 12],
        [ 1, 13],
        [ 2, 14],
        [ 3, 15]],

       [[ 4, 16],
        [ 5, 17],
        [ 6, 18],
        [ 7, 19]],

       [[ 8, 20],
        [ 9, 21],
        [10, 22],
        [11, 23]]])
>>> y = F.stack([x1, x2], axis=-1)
>>> y.shape
(3, 4, 2)
```

chainer.functions.swapaxes

chainer.functions.**swapaxes**(*x*, *axis1*, *axis2*)
Swap two axes of a variable.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **axis1** (*int*) – The first axis to swap.
- **axis2** (*int*) – The second axis to swap.

Returns Variable whose axes are swapped.

Return type *Variable*

Example

```
>>> x = np.array([[[0, 1, 2], [3, 4, 5]]], np.float32)
>>> x.shape
(1, 2, 3)
>>> y = F.swapaxes(x, axis1=0, axis2=1)
>>> y.shape
(2, 1, 3)
>>> y.array
array([[[0., 1., 2.]],

       [[3., 4., 5.]]], dtype=float32)
```

chainer.functions.tile

`chainer.functions.tile(x, reps)`

Construct an array by tiling a given array.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. Let the length of `reps` be `d`. If `x.ndim < d`, `x` is treated as `d`-dimensional array by prepending new axes. For example, when the shape of `x` is `(2,)` and tiled with 2-dim repetitions, `x` is treated as the shape `(1, 2)`. If `x.ndim > d`, `reps` is treated as `x.ndim`-dimensional by pre-pending 1's. For example, when the shape of `x` is `(2, 3, 2, 3)`, the 2-dim `reps` of `(2, 2)` is treated as `(1, 1, 2, 2)`.
- **reps** (`int` or `tuple` of `int` s) – The number of times which `x` is replicated along each axis.

Returns The tiled output Variable. Let the length of `reps` be `d`, the output has the dimension of `max(d, x.ndim)`.

Return type *Variable*

Example

```
>>> x = np.array([0, 1, 2])
>>> x.shape
(3,)
>>> y = F.tile(x, 2)
>>> y.shape
(6,)
>>> y.array
```

(continues on next page)

(continued from previous page)

```
array([0, 1, 2, 0, 1, 2])
>>> y = F.tile(x, (2, 2))
>>> y.shape
(2, 6)
>>> y.array
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> y = F.tile(x, (2, 1, 2))
>>> y.shape
(2, 1, 6)
>>> y.array
array([[[0, 1, 2, 0, 1, 2]],

       [[0, 1, 2, 0, 1, 2]]])
```

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x.shape
(2, 2)
>>> y = F.tile(x, 2)
>>> y.shape
(2, 4)
>>> y.array
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> y = F.tile(x, (2, 2))
>>> y.shape
(4, 4)
>>> y.array
array([[1, 2, 1, 2],
       [3, 4, 3, 4],
       [1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> y = F.tile(x, (2, 1, 2))
>>> y.shape
(2, 2, 4)
>>> y.array
array([[[1, 2, 1, 2],
        [3, 4, 3, 4]],

       [[1, 2, 1, 2],
        [3, 4, 3, 4]]])
```

chainer.functions.transpose

`chainer.functions.`**`transpose`**(*x*, *axes=None*)

Permute the dimensions of an input variable without copy.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable to be transposed. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **axes** (*tuple of ints*) – By default, reverse the dimensions, otherwise permute the axes according to the values given.

Returns Variable whose axes are permuted.

Return type *Variable*

Example

```
>>> x = np.array([[[0, 1, 2], [3, 4, 5]]], np.float32)
>>> x.shape
(1, 2, 3)
>>> y = F.transpose(x)  # reverse the dimensions
>>> y.shape
(3, 2, 1)
>>> y.array
array([[[0.],
        [3.]],

       [[1.],
        [4.]],

       [[2.],
        [5.]]], dtype=float32)
>>> y = F.transpose(x, axes=(1, 0, 2)) # swap 1st and 2nd axis
>>> y.shape
(2, 1, 3)
>>> y.array
array([[[0., 1., 2.]],

       [[3., 4., 5.]]], dtype=float32)
```

---

chainer.functions.transpose_sequence

`chainer.functions.transpose_sequence(xs)`

Transpose a list of Variables.

This function transposes a list of `Variable`s and returns a list of `Variables`. For example a user gives `[(0, 1, 2, 3), (4, 5), (6)]`, the function returns `[(0, 4, 6), (1, 5), (2), (3)]`. Note that a given list needs to be sorted by each length of `Variable`.

Parameters **xs** (list of `Variable` or *N-dimensional array*) – Variables to transpose.

Returns Transposed list.

Return type tuple of `Variable`

Example

```
>>> lst = [chainer.Variable(np.array([1, 1, 1])),
...        chainer.Variable(np.array([2, 2])),
...        chainer.Variable(np.array([3]))]
>>> lst
[variable([1, 1, 1]), variable([2, 2]), variable([3])]
>>> transposed = F.transpose_sequence(lst)
>>> transposed
(variable([1, 2, 3]), variable([1, 2]), variable([1]))
```

---

chainer.functions.vstack

`chainer.functions.vstack(xs)`

Concatenate variables vertically (row wise).

Parameters **xs** (list of `Variable` or *N-dimensional array*) – Input variables to be concatenated. The variables must have the same `ndim`. When the variables have the second axis (i.e. $ndim \geq 2$), the variables must have the same shape along all but the first axis. When the variables do not have the second axis(i.e. $ndim < 2$), the variables must have the same shape.

Returns Output variable. When the input variables have the second axis (i.e. $ndim \geq 2$), the shapes of inputs and output are the same along all but the first axis. The length of first axis is the sum of the lengths of inputs' first axis. When the variables do not have the second axis (i.e. $ndim < 2$), the shape of output is `(2, N)` (`N` is the size of the input variable).

Return type *Variable*

Example

```
>>> x1 = np.array((1, 2, 3))
>>> x1.shape
(3,)
>>> x2 = np.array((2, 3, 4))
>>> x2.shape
(3,)
>>> y = F.vstack((x1, x2))
>>> y.shape
(2, 3)
>>> y.array
array([[1, 2, 3],
       [2, 3, 4]])
>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> x2 = np.arange(12, 20).reshape(2, 4)
>>> x2.shape
(2, 4)
>>> x2
array([[12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> y = F.vstack([x1, x2])
>>> y.shape
(5, 4)
>>> y.array
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

chainer.functions.where

`chainer.functions.where(condition, x, y)`

Choose elements depending on condition.

This function choose values depending on a given `condition`. All `condition`, `x`, and `y` must have the same shape.

Parameters

- **condition** (*Variable* or *N-dimensional array*) – Input variable containing the condition. A $(s_1, s_2, ..., s_N)$ -shaped boolean array. Only boolean array is permitted.
- **x** (*Variable* or *N-dimensional array*) – Input variable chosen when `condition` is `True`. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **y** (*Variable* or *N-dimensional array*) – Input variable chosen when `condition` is `False`. A $(s_1, s_2, ..., s_N)$ -shaped float array.

Returns Variable containing chosen values.

Return type *Variable*

Example

```
>>> cond = np.array([[1, 0], [0, 1]], dtype=np.bool)
>>> cond
array([[ True, False],
       [False,  True]])
>>> x = np.array([[1, 2], [3, 4]], np.float32)
>>> y = np.zeros((2, 2), np.float32)
>>> F.where(cond, x, y).array
array([[1., 0.],
       [0., 4.]], dtype=float32)
```

4.2.4 Neural network connections

chainer.functions.bilinear	Applies a bilinear function based on given parameters.
chainer.functions.convolution_1d	1-dimensional convolution function.
chainer.functions.convolution_2d	Two-dimensional convolution function.
chainer.functions.convolution_3d	3-dimensional convolution function.
chainer.functions.convolution_nd	N-dimensional convolution function.
chainer.functions.deconvolution_1d	1-dimensional deconvolution function.
chainer.functions.deconvolution_2d	Two dimensional deconvolution function.
chainer.functions.deconvolution_3d	3-dimensional deconvolution function.
chainer.functions.deconvolution_nd	N-dimensional deconvolution function.
chainer.functions.depthwise_convolution_2d	Two-dimensional depthwise convolution function.
chainer.functions.deformable_convolution_2d_sampler	Two-dimensional deformable convolution function using computed offset.
chainer.functions.dilated_convolution_2d	Two-dimensional dilated convolution function.
chainer.functions.embed_id	Efficient linear function for one-hot input.

Continued on next page

Table 5 – continued from previous page

chainer.functions.linear	Linear function, or affine transformation.
chainer.functions.local_convolution_2d	Two-dimensional local convolution function.
chainer.functions.n_step_bigru	Stacked Bi-directional Gated Recurrent Unit function.
chainer.functions.n_step_bilstm	Stacked Bi-directional Long Short-Term Memory function.
chainer.functions.n_step_birnn	Stacked Bi-directional RNN function for sequence inputs.
chainer.functions.n_step_gru	Stacked Uni-directional Gated Recurrent Unit function.
chainer.functions.n_step_lstm	Stacked Uni-directional Long Short-Term Memory function.
chainer.functions.n_step_rnn	Stacked Uni-directional RNN function for sequence inputs.
chainer.functions.shift	Shift function.

chainer.functions.bilinear

`chainer.functions.bilinear(e1, e2, W, V1=None, V2=None, b=None)`

Applies a bilinear function based on given parameters.

This is a building block of Neural Tensor Network (see the reference paper below). It takes two input variables and one or four parameters, and outputs one variable.

To be precise, denote six input arrays mathematically by $e^1 \in \mathbb{R}^{I \cdot J}$, $e^2 \in \mathbb{R}^{I \cdot K}$, $W \in \mathbb{R}^{J \cdot K \cdot L}$, $V^1 \in \mathbb{R}^{J \cdot L}$, $V^2 \in \mathbb{R}^{K \cdot L}$, and $b \in \mathbb{R}^L$, where I is mini-batch size. In this document, we call V^1, V^2, and b linear parameters.

The output of forward propagation is calculated as

$$y_{il} = \sum_{jk} e^1_{ij} e^2_{ik} W_{jkl} + \sum_j e^1_{ij} V^1_{jl} + \sum_k e^2_{ik} V^2_{kl} + b_l.$$

Note that V1, V2, b are optional. If these are not given, then this function omits the last three terms in the above equation.

Note: This function accepts an input variable `e1` or `e2` of a non-matrix array. In this case, the leading dimension is treated as the batch dimension, and the other dimensions are reduced to one dimension.

Note: In the original paper, J and K must be equal and the author denotes $[V^1 V^2]$ (concatenation of matrices) by V.

Parameters

- **e1** (`Variable` or *N-dimensional array*) – Left input variable.
- **e2** (`Variable` or *N-dimensional array*) – Right input variable.
- **W** (`Variable` or *N-dimensional array*) – Quadratic weight variable.
- **V1** (`Variable` or *N-dimensional array*) – Left coefficient variable.
- **V2** (`Variable` or *N-dimensional array*) – Right coefficient variable.
- **b** (`Variable` or *N-dimensional array*) – Bias variable.

Returns Output variable.

Return type *Variable*

See: Reasoning With Neural Tensor Networks for Knowledge Base Completion [Socher+, NIPS2013].

See also:

`Bilinear` to manage the model parameters `W`, `V1`, `V2`, and `b`.

chainer.functions.convolution_1d

`chainer.functions.convolution_1d(x, W, b=None, stride=1, pad=0, cover_all=False, dilate=1, groups=1)`

1-dimensional convolution function.

Note: This function calls `convolution_nd()` internally, so see the details of the behavior in the documentation of `convolution_nd()`.

chainer.functions.convolution_2d

`chainer.functions.convolution_2d(x, W, b=None, stride=1, pad=0, cover_all=False, *, dilate=1, groups=1)`

Two-dimensional convolution function.

This is an implementation of two-dimensional convolution in ConvNets. It takes three variables: the input image `x`, the filter weight `W`, and the bias vector `b`.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I and c_O are the number of the input and output channels, respectively.
- h_I and w_I are the height and width of the input image, respectively.
- h_K and w_K are the height and width of the filters, respectively.
- h_P and w_P are the height and width of the spatial padding size, respectively.

Then the `Convolution2D` function computes correlations between filters and patches of size (h_K, w_K) in `x`. Note that correlation here is equivalent to the inner product between expanded vectors. Patches are extracted at positions shifted by multiples of `stride` from the first position `(-h_P, -w_P)` for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Let (s_Y, s_X) be the stride of filter application. Then, the output size (h_O, w_O) is determined by the following equations:

$$h_O = (h_I + 2h_P - h_K)/s_Y + 1,$$
$$w_O = (w_I + 2w_P - w_K)/s_X + 1.$$

If `cover_all` option is `True`, the filter will cover the all spatial locations. So, if the last stride of filter does not cover the end of spatial locations, an additional stride will be applied to the end part of spatial locations. In this case, the output size (h_O, w_O) is determined by the following equations:

$$h_O = (h_I + 2h_P - h_K + s_Y - 1)/s_Y + 1,$$
$$w_O = (w_I + 2w_P - w_K + s_X - 1)/s_X + 1.$$

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.cudnn_deterministic` is `True` and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

Convolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

When the dilation factor is greater than one, cuDNN is not used unless the version is 6.0 or higher.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape (n, c_I, h_I, w_I).
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape (c_O, c_I, h_K, w_K).
- **b** (None or `Variable` or *N-dimensional array*) – Bias variable of length c_O (optional).
- **stride** (`int` or pair of `int` s) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (`int` or pair of `int` s) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **cover_all** (`bool`) – If `True`, all spatial locations are convoluted into some output pixels.
- **dilate** (`int` or pair of `int` s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **groups** (`int`) – Number of groups of channels. If the number is greater than 1, input tensor W is divided into some blocks by this value. For each tensor blocks, convolution operation will be executed independently. Input channel size c_I and output channel size c_O must be exactly divisible by this value.

Returns Output variable of shape (n, c_O, h_O, w_O).

Return type *Variable*

See also:

`Convolution2D` to manage the model parameters `W` and `b`.

Example

```
>>> n = 10
>>> c_i, c_o = 3, 1
>>> h_i, w_i = 30, 40
>>> h_k, w_k = 10, 10
>>> h_p, w_p = 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, h_i, w_i)).astype(np.float32)
>>> x.shape
(10, 3, 30, 40)
>>> W = np.random.uniform(0, 1, (c_o, c_i, h_k, w_k)).astype(np.float32)
>>> W.shape
(1, 3, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o,)).astype(np.float32)
>>> b.shape
(1,)
>>> s_y, s_x = 5, 7
>>> y = F.convolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p))
>>> y.shape
(10, 1, 7, 6)
```

(continues on next page)

(continued from previous page)

```
>>> h_o = int((h_i + 2 * h_p - h_k) / s_y + 1)
>>> w_o = int((w_i + 2 * w_p - w_k) / s_x + 1)
>>> y.shape == (n, c_o, h_o, w_o)
True
>>> y = F.convolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p), cover_
↪all=True)
>>> y.shape == (n, c_o, h_o, w_o + 1)
True
```

chainer.functions.convolution_3d

`chainer.functions.convolution_3d(x, W, b=None, stride=1, pad=0, cover_all=False, dilate=1, groups=1)`

3-dimensional convolution function.

Note: This function calls `convolution_nd()` internally, so see the details of the behavior in the documentation of `convolution_nd()`.

chainer.functions.convolution_nd

`chainer.functions.convolution_nd(x, W, b=None, stride=1, pad=0, cover_all=False, dilate=1, groups=1)`

N-dimensional convolution function.

This is an implementation of N-dimensional convolution which is generalized two-dimensional convolution in ConvNets. It takes three variables: the input `x`, the filter weight `W` and the bias vector `b`.

Notation: here is a notation for dimensionalities.

- N is the number of spatial dimensions.
- n is the batch size.
- c_I and c_O are the number of the input and output channels, respectively.
- $d_1, d_2, ..., d_N$ are the size of each axis of the input's spatial dimensions, respectively.
- $k_1, k_2, ..., k_N$ are the size of each axis of the filters, respectively.
- $l_1, l_2, ..., l_N$ are the size of each axis of the output's spatial dimensions, respectively.
- $p_1, p_2, ..., p_N$ are the size of each axis of the spatial padding size, respectively.

Then the `convolution_nd` function computes correlations between filters and patches of size $(k_1, k_2, ..., k_N)$ in `x`. Note that correlation here is equivalent to the inner product between expanded tensors. Patches are extracted at positions shifted by multiples of `stride` from the first position `(-p_1, -p_2, ..., -p_N)` for each spatial axis.

Let $(s_1, s_2, ..., s_N)$ be the stride of filter application. Then, the output size $(l_1, l_2, ..., l_N)$ is determined by the following equations:

$$l_n = (d_n + 2p_n - k_n)/s_n + 1 \ \ (n = 1, ..., N)$$

If `cover_all` option is `True`, the filter will cover the all spatial locations. So, if the last stride of filter does not cover the end of spatial locations, an additional stride will be applied to the end part of spatial locations. In this case, the output size is determined by the following equations:

$$l_n = (d_n + 2p_n - k_n + s_n - 1)/s_n + 1 \ \ (n = 1, ..., N)$$

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape $(n, c_I, d_1, d_2, ..., d_N)$.
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape $(c_O, c_I, k_1, k_2, ..., k_N)$.
- **b** (None or `Variable` or *N-dimensional array*) – One-dimensional bias variable with length c_O (optional).
- **stride** (`int` or `tuple` of `int` s) – Stride of filter applications $(s_1, s_2, ..., s_N)$. `stride=s` is equivalent to `(s, s, ..., s)`.
- **pad** (`int` or `tuple` of `int` s) – Spatial padding width for input arrays $(p_1, p_2, ..., p_N)$. `pad=p` is equivalent to `(p, p, ..., p)`.
- **cover_all** (*bool*) – If `True`, all spatial locations are convoluted into some output pixels. It may make the output size larger. *cover_all* needs to be `False` if you want to use cuDNN.
- **dilate** (`int` or `tuple` of `int` s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d, ..., d)` are equivalent.
- **groups** (`int`) – The number of groups to use grouped convolution. The default is one, where grouped convolution is not used.

Returns Output variable of shape $(n, c_O, l_1, l_2, ..., l_N)$.

Return type *Variable*

Note: This function uses cuDNN implementation for its forward and backward computation if ALL of the following conditions are satisfied:

- `cuda.cudnn_enabled` is `True`
- `chainer.config.use_cudnn` is `'always'` or `'auto'`
- The number of spatial dimensions is more than one.
- `cover_all` is `False`
- The input's `dtype` is equal to the filter weight's.
- The `dtype` is FP16, FP32 or FP64. (FP16 is only available when cuDNN version $\geq$ v3.)

Convolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

See also:

`ConvolutionND` to manage the model parameters `W` and `b`.

See also:

`convolution_2d()`

Example

```
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 30, 40, 50
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype(np.float32)
>>> x.shape
(10, 3, 30, 40, 50)
>>> W = np.random.uniform(0, 1, (c_o, c_i, k1, k2, k3)).astype(np.float32)
>>> W.shape
(1, 3, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype(np.float32)
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> y = F.convolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3))
>>> y.shape
(10, 1, 16, 11, 9)
>>> l1 = int((d1 + 2 * p1 - k1) / s1 + 1)
>>> l2 = int((d2 + 2 * p2 - k2) / s2 + 1)
>>> l3 = int((d3 + 2 * p3 - k3) / s3 + 1)
>>> y.shape == (n, c_o, l1, l2, l3)
True
>>> y = F.convolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3), cover_
↪all=True)
>>> y.shape == (n, c_o, l1, l2, l3 + 1)
True
```

chainer.functions.deconvolution_1d

`chainer.functions.deconvolution_1d(x, W, b=None, stride=1, pad=0, outsize=None, dilate=1, groups=1)`

1-dimensional deconvolution function.

Note: This function calls `deconvolution_nd()` internally, so see the details of the behavior in the documentation of `deconvolution_nd()`.

chainer.functions.deconvolution_2d

`chainer.functions.deconvolution_2d(x, W, b=None, stride=1, pad=0, outsize=None, *, dilate=1, groups=1)`

Two dimensional deconvolution function.

This is an implementation of two-dimensional deconvolution. In most of deep learning frameworks and papers, this function is called **transposed convolution**. But because of historical reasons (e.g. paper by Ziller Deconvolutional Networks) and backward compatibility, this function is called **deconvolution** in Chainer.

It takes three variables: input image `x`, the filter weight `W`, and the bias vector `b`.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I and c_O are the number of the input and output channels, respectively.

- h_I and w_I are the height and width of the input image, respectively.
- h_K and w_K are the height and width of the filters, respectively.
- h_P and w_P are the height and width of the spatial padding size, respectively.

Let (s_Y, s_X) be the stride of filter application. Then, the output size (h_O, w_O) is estimated by the following equations:

$$h_O = s_Y(h_I - 1) + h_K - 2h_P,$$
$$w_O = s_X(w_I - 1) + w_K - 2w_P.$$

The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.deterministic` is `True` and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

Deconvolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape (n, c_I, h_I, w_I).
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape (c_I, c_O, h_K, w_K).
- **b** (None or `Variable` or *N-dimensional array*) – Bias variable of length c_O (optional).
- **stride** (`int` or pair of `int` s) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (`int` or pair of `int` s) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **outsize** (None or `tuple` of `int` s) – Expected output size of deconvolutional operation. It should be pair of height and width (h_O, w_O). Default value is `None` and the outsize is estimated by input size, stride and pad.
- **dilate** (`int` or pair of `int` s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **groups** (`int`) – The number of groups to use grouped deconvolution. The default is one, where grouped deconvolution is not used.

Returns Output variable of shape (n, c_O, h_O, w_O).

Return type *Variable*

See also:

`Deconvolution2D` to manage the model parameters `W` and `b`.

Example

```
>>> n = 10
>>> c_i, c_o = 1, 3
>>> h_i, w_i = 5, 10
>>> h_k, w_k = 10, 10
>>> h_p, w_p = 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, h_i, w_i)).astype(np.float32)
>>> x.shape
(10, 1, 5, 10)
```

(continues on next page)

(continued from previous page)

```
>>> W = np.random.uniform(0, 1, (c_i, c_o, h_k, w_k)).astype(np.float32)
>>> W.shape
(1, 3, 10, 10)
>>> b = np.random.uniform(0, 1, c_o).astype(np.float32)
>>> b.shape
(3,)
>>> s_y, s_x = 5, 5
>>> y = F.deconvolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p))
>>> y.shape
(10, 3, 20, 45)
>>> h_o = s_y * (h_i - 1) + h_k - 2 * h_p
>>> w_o = s_x * (w_i - 1) + w_k - 2 * w_p
>>> y.shape == (n, c_o, h_o, w_o)
True
```

chainer.functions.deconvolution_3d

`chainer.functions.`**`deconvolution_3d`**(*x*, *W*, *b=None*, *stride=1*, *pad=0*, *outsize=None*, *dilate=1*, *groups=1*)

3-dimensional deconvolution function.

Note: This function calls `deconvolution_nd()` internally, so see the details of the behavior in the documentation of `deconvolution_nd()`.

chainer.functions.deconvolution_nd

`chainer.functions.`**`deconvolution_nd`**(*x*, *W*, *b=None*, *stride=1*, *pad=0*, *outsize=None*, *dilate=1*, *groups=1*)

N-dimensional deconvolution function.

This is an implementation of N-dimensional deconvolution which generalizes two-dimensional one. In most of deep learning frameworks and papers, this function is called **transposed convolution**. But because of historical reasons (e.g. paper by Ziller Deconvolutional Networks) and backward compatibility, this function is called **deconvolution** in Chainer.

It takes three variables: the input `x`, the filter weight `W`, and the bias vector `b`.

Notation: here is a notation for dimensionalities.

- N is the number of spatial dimensions.
- n is the batch size.
- c_I and c_O are the number of the input and output channels, respectively.
- $d_1, d_2, ..., d_N$ are the size of each axis of the input's spatial dimensions, respectively.
- $k_1, k_2, ..., k_N$ are the size of each axis of the filters, respectively.
- $p_1, p_2, ..., p_N$ are the size of each axis of the spatial padding size, respectively.
- $s_1, s_2, ..., s_N$ are the stride of each axis of filter application, respectively.

If `outsize` option is `None`, the output size $(l_1, l_2, ..., l_N)$ is determined by the following equations with the items in the above list:

$$l_n = s_n(d_n - 1) + k_n - 2p_n \ \ (n = 1, ..., N)$$

If `outsize` option is given, the output size is determined by `outsize`. In this case, the `outsize` $(l_1, l_2, ..., l_N)$ must satisfy the following equations:

$$d_n = \lfloor (l_n + 2p_n - k_n)/s_n \rfloor + 1 \ \ (n = 1, ..., N)$$

Deconvolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape $(n, c_I, d_1, d_2, ..., d_N)$.
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape $(c_I, c_O, k_1, k_2, ..., k_N)$.
- **b** (None or `Variable` or *N-dimensional array*) – One-dimensional bias variable with length c_O (optional).
- **stride** (`int` or `tuple` of `int` s) – Stride of filter applications $(s_1, s_2, ..., s_N)$. `stride=s` is equivalent to `(s, s, ..., s)`.
- **pad** (`int` or `tuple` of `int` s) – Spatial padding width for input arrays $(p_1, p_2, ..., p_N)$. `pad=p` is equivalent to `(p, p, ..., p)`.
- **outsize** (None or `tuple` of `int` s) – Expected output size of deconvolutional operation. It should be a tuple of ints $(l_1, l_2, ..., l_N)$. Default value is `None` and the outsize is estimated by input size, stride and pad.
- **dilate** (`int` or `tuple` of `int` s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d, ..., d)` are equivalent.
- **groups** (`int`) – The number of groups to use grouped convolution. The default is one, where grouped convolution is not used.

Returns Output variable of shape $(n, c_O, l_1, l_2, ..., l_N)$.

Return type *Variable*

See also:

`DeconvolutionND` to manage the model parameters `W` and `b`.

See also:

`deconvolution_2d()`

Example

Example1: the case when `outsize` is not given.

```
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 5, 10, 15
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype(np.float32)
>>> x.shape
```

(continues on next page)

(continued from previous page)

```
(10, 3, 5, 10, 15)
>>> W = np.random.uniform(0, 1, (c_i, c_o, k1, k2, k3)).astype(np.float32)
>>> W.shape
(3, 1, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype(np.float32)
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> y = F.deconvolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3))
>>> y.shape
(10, 1, 8, 36, 84)
>>> l1 = s1 * (d1 - 1) + k1 - 2 * p1
>>> l2 = s2 * (d2 - 1) + k2 - 2 * p2
>>> l3 = s3 * (d3 - 1) + k3 - 2 * p3
>>> y.shape == (n, c_o, l1, l2, l3)
True
```

Example2: the case when `outsize` is given.

```
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 5, 10, 15
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype(np.float32)
>>> x.shape
(10, 3, 5, 10, 15)
>>> W = np.random.uniform(0, 1, (c_i, c_o, k1, k2, k3)).astype(np.float32)
>>> W.shape
(3, 1, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype(np.float32)
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> l1, l2, l3 = 9, 38, 87
>>> d1 == int((l1 + 2 * p1 - k1) / s1) + 1
True
>>> d2 == int((l2 + 2 * p2 - k2) / s2) + 1
True
>>> d3 == int((l3 + 2 * p3 - k3) / s3) + 1
True
>>> y = F.deconvolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3),
↪outsize=(l1, l2, l3))
>>> y.shape
(10, 1, 9, 38, 87)
>>> y.shape == (n, c_o, l1, l2, l3)
True
```

chainer.functions.depthwise_convolution_2d

`chainer.functions.depthwise_convolution_2d`(*x*, *W*, *b=None*, *stride=1*, *pad=0*)

Two-dimensional depthwise convolution function.

This is an implementation of two-dimensional depthwise convolution. It takes two or three variables: the input image `x`, the filter weight `W`, and optionally, the bias vector `b`.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I is the number of the input.
- c_M is the channel multiplier.
- h and w are the height and width of the input image, respectively.
- h_O and w_O are the height and width of the output image, respectively.
- k_H and k_W are the height and width of the filters, respectively.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **W** (*Variable* or *N-dimensional array*) – Weight variable of shape (c_M, c_I, k_H, k_W).
- **b** (*Variable* or *N-dimensional array*) – Bias variable of length $c_M * c_I$ (optional).
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.

Returns Output variable. Its shape is $(n, c_I * c_M, h_O, w_O)$.

Return type *Variable*

Like `Convolution2D`, `DepthwiseConvolution2D` function computes correlations between filters and patches of size (k_H, k_W) in `x`. But unlike `Convolution2D`, `DepthwiseConvolution2D` does not add up input channels of filters but concatenates them. For that reason, the shape of outputs of depthwise convolution are $(n, c_I * c_M, h_O, w_O)$, c_M is called channel_multiplier.

(h_O, w_O) is determined by the equivalent equation of `Convolution2D`.

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

See: L. Sifre. Rigid-motion scattering for image classification

See also:

DepthwiseConvolution2D to manage the model parameters `W` and `b`.

Example

```
>>> x = np.random.uniform(0, 1, (2, 3, 4, 7))
>>> W = np.random.uniform(0, 1, (2, 3, 3, 3))
>>> b = np.random.uniform(0, 1, (6,))
>>> y = F.depthwise_convolution_2d(x, W, b)
>>> y.shape
(2, 6, 2, 5)
```

chainer.functions.deformable_convolution_2d_sampler

`chainer.functions.`**`deformable_convolution_2d_sampler`**(*x, offset, W, b=None, stride=1, pad=0*)

Two-dimensional deformable convolution function using computed offset.

This is an implementation of two-dimensional deformable convolution from Deformable Convolutional Networks.

It takes four variables: the input image `x`, the offset image `offset`, the filter weight `W`, and the bias vector `b`.

Notation: here is the notation for the dimensionalities.

- n is the batch size.
- c_I and c_O are the number of the input and output, respectively.
- h and w are the height and width of the input image, respectively.
- k_H and k_W are the height and width of the filters, respectively.
- s_Y and s_X are the strides of the filter.
- p_H and p_W are the spatial padding sizes.

The output size (h_O, w_O) is determined by the following equations:

$$h_O = (h + 2p_H - k_H)/s_Y + 1,$$
$$w_O = (w + 2p_W - k_W)/s_X + 1.$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **offset** (*Variable* or *N-dimensional array*) – Offset variable of shape $(n, 2 \cdot k_H \cdot k_W, h_O, w_O)$. The first $k_H \cdot k_W$ index of the second axis corresponds to the offsets in the horizontal direction. The last $k_H \cdot k_W$ index of the second axis corresponds to the offsets in the vertical direction.
- **W** (*Variable* or *N-dimensional array*) – Weight variable of shape (c_O, c_I, k_H, k_W).
- **b** (*Variable* or *N-dimensional array*) – Bias variable of length c_O (optional).
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.

Returns Output variable.

Return type *Variable*

Deformable convolution adds 2D offsets to the regular grid sampling locations in the standard convolution. It enables free form deformation of the sampling grid.

See Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, Yichen Wei. Deformable Convolutional Networks

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

See also:

DeformableConvolution2D to manage the model parameters `W` and `b`.

Example

```
>>> x = np.random.uniform(0, 1, (2, 3, 4, 7)).astype(np.float32)
>>> offset = np.random.uniform(
...     0, 1, (2, 2 * 3 * 3, 2, 5)).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
>>> W = np.random.uniform(0, 1, (4, 3, 3, 3)).astype(np.float32)
>>> b = np.random.uniform(0, 1, (4,)).astype(np.float32)
>>> y = F.deformable_convolution_2d_sampler(x, offset, W, b)
>>> y.shape
(2, 4, 2, 5)
```

chainer.functions.dilated_convolution_2d

`chainer.functions.dilated_convolution_2d`(*x*, *W*, *b=None*, *stride=1*, *pad=0*, *dilate=1*, *cover_all=False*)

Two-dimensional dilated convolution function.

This is an implementation of two-dimensional dilated convolution in ConvNets. It takes three variables: the input image `x`, the filter weight `W`, and the bias vector `b`.

Note: You can also perform dilated convolution by passing `dilate` argument to `chainer.functions.convolution_2d`. The functionality is the same.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I and c_O are the number of the input and output, respectively.
- h and w are the height and width of the input image, respectively.
- k_H and k_W are the height and width of the filters, respectively.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape (c_O, c_I, k_H, k_W).
- **b** (`Variable` or *N-dimensional array*) – Bias variable of length c_O (optional).
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **dilate** (*int or pair of ints*) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **cover_all** (*bool*) – If `True`, all spatial locations are convoluted into some output pixels. It may make the output size larger.

Returns Output variable.

Return type *Variable*

The two-dimensional dilated convolution function is defined as follows. Then the `DilatedConvolution2D` function computes correlations between filters and patches of size (k_H, k_W) in `x`. Patches here are extracted at intervals of the dilation factor. Note that correlation here is equivalent to the inner product between expanded vectors. Patches are extracted at intervals of the dilation factor and at positions shifted by multiples of `stride` from the first position -pad for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Let (s_Y, s_X) be the stride of filter application, (p_H, p_W) the spatial padding size, and (d_Y, d_X) the dilation factor of filter application. Then, the output size (h_O, w_O) is determined by the following equations:

$$h_O = (h + 2p_H - k_H - (k_H - 1) * (d_Y - 1))/s_Y + 1,$$
$$w_O = (w + 2p_W - k_W - (k_W - 1) * (d_X - 1))/s_X + 1.$$

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

chainer.functions.embed_id

chainer.functions.**embed_id**(*x*, *W*, *ignore_label=None*)

Efficient linear function for one-hot input.

This function implements so called *word embeddings*. It takes two arguments: a set of IDs (words) x in B dimensional integer vector, and a set of all ID (word) embeddings W in $V \times d$ float matrix. It outputs $B \times d$ matrix whose i-th row is the x[i]-th row of W.

This function is only differentiable on the input W.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Batch vectors of IDs. Each element must be signed integer.
- **W** (*Variable* or *N-dimensional array*) – Distributed representation of each ID (a.k.a. word embeddings).
- **ignore_label** (int or None) – If ignore_label is an int value, i-th row of return value is filled with 0.

Returns Output variable.

Return type *Variable*

See also:

EmbedID to manage the model parameter W.

Example

```
>>> x = np.array([2, 1]).astype(np.int32)
>>> x
array([2, 1], dtype=int32)
>>> W = np.array([[0, 0, 0],
...               [1, 1, 1],
...               [2, 2, 2]]).astype(np.float32)
>>> W
array([[0., 0., 0.],
       [1., 1., 1.],
       [2., 2., 2.]], dtype=float32)
>>> F.embed_id(x, W).array
array([[2., 2., 2.],
       [1., 1., 1.]], dtype=float32)
>>> F.embed_id(x, W, ignore_label=1).array
array([[2., 2., 2.],
       [0., 0., 0.]], dtype=float32)
```

chainer.functions.linear

`chainer.functions.linear(x, W, b=None, n_batch_axes=1)`

Linear function, or affine transformation.

It accepts two or three arguments: an input minibatch `x`, a weight matrix `W`, and optionally a bias vector `b`. It computes

$$y_i = W x_i + b.$$

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable, which is a $(s_1, s_2, ..., s_n)$-shaped float array. Its first `n_batch_axes` dimensions are handled as *minibatch dimensions*. The other dimensions are handled as concatenated one dimension whose size must be $(s_{\text{n\_batch\_axes}} * ... * s_n = N)$.
- **W** (`Variable` or *N-dimensional array*) – Weight variable of shape (M, N), where $(N = s_{\text{n\_batch\_axes}} * ... * s_n)$.
- **b** (`Variable` or *N-dimensional array*) – Bias variable (optional) of shape $(M,)$.
- **n_batch_axes** (*int*) – The number of batch axes. The default is 1. The input variable is reshaped into $(\text{n\_batch\_axes} + 1)$-dimensional tensor. This should be greater than 0.

Returns Output variable. A float array with shape of $(s_1, ..., s_{\text{n\_batch\_axes}}, M)$.

Return type *Variable*

See also:

`Linear` to manage the model parameters `W` and `b`.

Example

```
>>> x = np.random.uniform(0, 1, (3, 4)).astype(np.float32)
>>> W = np.random.uniform(0, 1, (5, 4)).astype(np.float32)
>>> b = np.random.uniform(0, 1, (5,)).astype(np.float32)
>>> y = F.linear(x, W, b)
>>> y.shape
(3, 5)
```

chainer.functions.local_convolution_2d

`chainer.functions.local_convolution_2d(x, W, b=None, stride=1)`

Two-dimensional local convolution function.

Locally-connected function for 2D inputs. Works similarly to convolution_2d, except that weights are unshared, that is, a different set of filters is applied at each different patch of the input. It takes two or three variables: the input image `x`, the filter weight `W`, and optionally, the bias vector `b`.

Notation: here is a notation for dimensionalities.

- n is the batch size.
- c_I is the number of the input.
- c_O is the number of output channels.
- h and w are the height and width of the input image, respectively.

- h_O and w_O are the height and width of the output image, respectively.
- k_H and k_W are the height and width of the filters, respectively.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape (n, c_I, h, w).
- **W** (*Variable* or *N-dimensional array*) – Weight variable of shape $(c_O, h_O, w_O, c_I, k_H, k_W)$.
- **b** (*Variable* or *N-dimensional array*) – Bias variable of shape (c_O, h_O, w_O) (optional).
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.

Returns Output variable. Its shape is $(n, c_I * c_O, h_O, w_O)$.

Return type *Variable*

Like `Convolution2D`, `LocalConvolution2D` function computes correlations between filters and patches of size (k_H, k_W) in `x`. But unlike `Convolution2D`, `LocalConvolution2D` has a separate filter for each patch of the input

(h_O, w_O) is determined by the equivalent equation of `Convolution2D`, without any padding

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

See also:

LocalConvolution2D to manage the model parameters `W` and `b`.

Example

```
>>> x = np.random.uniform(0, 1, (2, 3, 7, 7))
>>> W = np.random.uniform(0, 1, (2, 5, 5, 3, 3, 3))
>>> b = np.random.uniform(0, 1, (2, 5, 5))
>>> y = F.local_convolution_2d(x, W, b)
>>> y.shape
(2, 2, 5, 5)
```

---

chainer.functions.n_step_bigru

`chainer.functions.`**`n_step_bigru`**(*n_layers, dropout_ratio, hx, ws, bs, xs*)

Stacked Bi-directional Gated Recurrent Unit function.

This function calculates stacked Bi-directional GRU with sequences. This function gets an initial hidden state h_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t for each time t from input x_t.

$$
\begin{aligned}
r_t^f &= \sigma(W_0^f x_t + W_3^f h_{t-1} + b_0^f + b_3^f) \\
z_t^f &= \sigma(W_1^f x_t + W_4^f h_{t-1} + b_1^f + b_4^f) \\
h_t^{f'} &= \tanh(W_2^f x_t + b_2^f + r_t^f \cdot (W_5^f h_{t-1} + b_5^f)) \\
h_t^f &= (1 - z_t^f) \cdot h_t^{f'} + z_t^f \cdot h_{t-1} \\
r_t^b &= \sigma(W_0^b x_t + W_3^b h_{t-1} + b_0^b + b_3^b) \\
z_t^b &= \sigma(W_1^b x_t + W_4^b h_{t-1} + b_1^b + b_4^b) \\
h_t^{b'} &= \tanh(W_2^b x_t + b_2^b + r_t^b \cdot (W_5^b h_{t-1} + b_5^b)) \\
h_t^b &= (1 - z_t^b) \cdot h_t^{b'} + z_t^b \cdot h_{t-1} \\
h_t &= [h_t^f; h_t^b]
\end{aligned}
$$

where W^f is weight matrices for forward-GRU, W^b is weight matrices for backward-GRU.

As the function accepts a sequence, it calculates h_t for all t with one call. Six weight matrices and six bias vectors are required for each layers. So, when S layers exists, you need to prepare $6S$ weight matrices and $6S$ bias vectors.

If the number of layers `n_layers` is greather than 1, input of `k`-th layer is hidden state `h_t` of `k-1`-th layer. Note that all input variables except first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – Number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*Variable*) – Variable holding stacked hidden states. Its shape is `(2S, B, N)` where `S` is number of layers and is equal to `n_layers`, `B` is mini-batch size, and `N` is dimension of hidden units.
- **ws** (list of list of *Variable*) – Weight matrices. `ws[i]` represents weights for i-th layer. Each `ws[i]` is a list containing six matrices. `ws[i][j]` is corresponding with `W_j` in the equation. Only `ws[0][j]` where `0 <= j < 3` is `(I, N)` shape as they are multiplied with input variables. All other matrices has `(N, N)` shape.
- **bs** (list of list of *Variable*) – Bias vectors. `bs[i]` represnents biases for i-th layer. Each `bs[i]` is a list containing six vectors. `bs[i][j]` is corresponding with `b_j` in the equation. Shape of each matrix is `(N,)` where `N` is dimension of hidden units.
- **xs** (list of *Variable*) – A list of *Variable* holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is mini-batch size for time `t`, and `I` is size of input units. Note that this function supports variable length sequences. When sequneces has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. `transpose_sequence()` transpose a list of `Variable()` holding sequence. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.
- **use_bi_direction** (*bool*) – If `True`, this function uses Bi-direction GRU.

Returns

This function returns a tuple containing three elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.

- `ys` is a list of *Variable*. Each element `ys[t]` holds hidden states of the last layer corresponding to an input `xs[t]`. Its shape is `(B_t, N)` where `B_t` is mini-batch size for time `t`, and `N` is size of hidden units. Note that `B_t` is the same value as `xs[t]`.

Return type tuple

chainer.functions.n_step_bilstm

`chainer.functions.n_step_bilstm`(*n_layers*, *dropout_ratio*, *hx*, *cx*, *ws*, *bs*, *xs*)

Stacked Bi-directional Long Short-Term Memory function.

This function calculates stacked Bi-directional LSTM with sequences. This function gets an initial hidden state h_0, an initial cell state c_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t and c_t for each time t from input x_t.

$$
\begin{aligned}
i^f_t &= \sigma(W^f_0 x_t + W^f_4 h_{t-1} + b^f_0 + b^f_4), \\
f^f_t &= \sigma(W^f_1 x_t + W^f_5 h_{t-1} + b^f_1 + b^f_5), \\
o^f_t &= \sigma(W^f_2 x_t + W^f_6 h_{t-1} + b^f_2 + b^f_6), \\
a^f_t &= \tanh(W^f_3 x_t + W^f_7 h_{t-1} + b^f_3 + b^f_7), \\
c^f_t &= f^f_t \cdot c^f_{t-1} + i^f_t \cdot a^f_t, \\
h^f_t &= o^f_t \cdot \tanh(c^f_t), \\
i^b_t &= \sigma(W^b_0 x_t + W^b_4 h_{t-1} + b^b_0 + b^b_4), \\
f^b_t &= \sigma(W^b_1 x_t + W^b_5 h_{t-1} + b^b_1 + b^b_5), \\
o^b_t &= \sigma(W^b_2 x_t + W^b_6 h_{t-1} + b^b_2 + b^b_6), \\
a^b_t &= \tanh(W^b_3 x_t + W^b_7 h_{t-1} + b^b_3 + b^b_7), \\
c^b_t &= f^b_t \cdot c^b_{t-1} + i^b_t \cdot a^b_t, \\
h^b_t &= o^b_t \cdot \tanh(c^b_t), \\
h_t &= [h^f_t; h^b_t]
\end{aligned}
$$

where W^f is the weight matrices for forward-LSTM, W^b is weight matrices for backward-LSTM.

As the function accepts a sequence, it calculates h_t for all t with one call. Eight weight matrices and eight bias vectors are required for each layer of each direction. So, when S layers exist, you need to prepare $16S$ weight matrices and $16S$ bias vectors.

If the number of layers `n_layers` is greater than 1, the input of the `k`-th layer is the hidden state `h_t` of the `k-1`-th layer. Note that all input variables except the first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – The number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*Variable*) – Variable holding stacked hidden states. Its shape is `(2S, B, N)` where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units. Because of bi-direction, the first dimension length is `2S`.
- **cx** (*Variable*) – Variable holding stacked cell states. It has the same shape as `hx`.
- **ws** (list of list of *Variable*) – Weight matrices. `ws[2 * l + m]` represents the weights for the l-th layer of the m-th direction. (`m == 0` means the forward direction and `m == 1` means the backward direction.) Each `ws[i]` is a list containing eight matrices. `ws[i][j]` corresponds to W_j in the equation. `ws[0][j]` and `ws[1][j]` where `0 <= j < 4` are `(I, N)`-shaped because they are multiplied with input variables, where `I` is the size of the input. `ws[i][j]` where `2 <= i` and `0 <= j < 4` are `(N, 2N)`-shaped because they are multiplied with two hidden layers $h_t = [h_t^f; h_t^b]$. All other matrices are `(N, N)`-shaped.
- **bs** (list of list of *Variable*) – Bias vectors. `bs[2 * l + m]` represents the weights for the l-th layer of m-th direction. (`m == 0` means the forward direction and `m == 1` means the backward direction.) Each `bs[i]` is a list containing eight vectors. `bs[i][j]` corresponds to b_j in the equation. The shape of each matrix is `(N,)`.
- **xs** (list of *Variable*) – A list of *Variable* holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is the mini-batch size for time `t`. The sequences must be transposed. *transpose_sequence()* can be used to transpose a list of *Variable*s each representing a sequence. When sequences has different lengths, they must be sorted in descending order of their lengths before transposing. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.

Returns

This function returns a tuple containing three elements, `hy`, `cy` and `ys`.

- `hy` is an updated hidden states whose shape is the same as `hx`.
- `cy` is an updated cell states whose shape is the same as `cx`.
- `ys` is a list of *Variable* . Each element `ys[t]` holds hidden states of the last layer corresponding to an input `xs[t]`. Its shape is `(B_t, 2N)` where `B_t` is the mini-batch size for time `t`, and `N` is size of hidden units. Note that `B_t` is the same value as `xs[t]`.

Return type tuple

Example

```
>>> batchs = [3, 2, 1]  # support variable length sequences
>>> in_size, out_size, n_layers = 3, 2, 2
>>> dropout_ratio = 0.0
>>> xs = [np.ones((b, in_size)).astype(np.float32) for b in batchs]
>>> [x.shape for x in xs]
[(3, 3), (2, 3), (1, 3)]
>>> h_shape = (n_layers * 2, batchs[0], out_size)
```

(continues on next page)

(continued from previous page)

```
>>> hx = np.ones(h_shape).astype(np.float32)
>>> cx = np.ones(h_shape).astype(np.float32)
>>> def w_in(i, j):
...     if i == 0 and j < 4:
...         return in_size
...     elif i > 0 and j < 4:
...         return out_size * 2
...     else:
...         return out_size
...
>>> ws = []
>>> bs = []
>>> for n in range(n_layers):
...     for direction in (0, 1):
...         ws.append([np.ones((out_size, w_in(n, i))).astype(np.float32) for i in range(8)])
...         bs.append([np.ones((out_size,)).astype(np.float32) for _ in range(8)])
...
>>> ws[0][0].shape  # ws[0:2][:4].shape are (out_size, in_size)
(2, 3)
>>> ws[2][0].shape  # ws[2:][:4].shape are (out_size, 2 * out_size)
(2, 4)
>>> ws[0][4].shape  # others are (out_size, out_size)
(2, 2)
>>> bs[0][0].shape
(2,)
>>> hy, cy, ys = F.n_step_bilstm(
...     n_layers, dropout_ratio, hx, cx, ws, bs, xs)
>>> hy.shape
(4, 3, 2)
>>> cy.shape
(4, 3, 2)
>>> [y.shape for y in ys]
[(3, 4), (2, 4), (1, 4)]
```

chainer.functions.n_step_birnn

chainer.functions.**n_step_birnn**(*n_layers, dropout_ratio, hx, ws, bs, xs, activation='tanh'*)

Stacked Bi-directional RNN function for sequence inputs.

This function calculates stacked Bi-directional RNN with sequences. This function gets an initial hidden state h_0, an initial cell state c_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t and c_t for each time t from input x_t.

$$
\begin{aligned}
h^f_t &= f(W^f_0 x_t + W^f_1 h_{t-1} + b^f_0 + b^f_1), \\
h^b_t &= f(W^b_0 x_t + W^b_1 h_{t-1} + b^b_0 + b^b_1), \\
h_t &= [h^f_t; h^f_t],
\end{aligned}
$$

where f is an activation function.

Weight matrices W contains two matrices W^f and W^b. W^f is weight matrices for forward directional RNN. W^b is weight matrices for backward directional RNN.

W^f contains W^f_0 for an input sequence and W^f_1 for a hidden state. W^b contains W^b_0 for an input sequence and W^b_1 for a hidden state.

Bias matrices b contains two matrices b^f and b^f. b^f contains b^f_0 for an input sequence and b^f_1 for a hidden state. b^b contains b^b_0 for an input sequence and b^b_1 for a hidden state.

As the function accepts a sequence, it calculates h_t for all t with one call. Two weight matrices and two bias vectors are required for each layer. So, when S layers exist, you need to prepare $2S$ weight matrices and $2S$ bias vectors.

If the number of layers `n_layers` is greather than 1, input of `k`-th layer is hidden state `h_t` of `k-1`-th layer. Note that all input variables except first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – Number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*Variable*) – Variable holding stacked hidden states. Its shape is `(2S, B, N)` where `S` is number of layers and is equal to `n_layers`, `B` is mini-batch size, and `N` is dimension of hidden units. Because of bi-direction, the first dimension length is `2S`.
- **ws** (list of list of *Variable*) – Weight matrices. `ws[i + di]` represents weights for i-th layer. Note that `di = 0` for forward-RNN and `di = 1` for backward-RNN. Each `ws[i + di]` is a list containing two matrices. `ws[i + di][j]` is corresponding with `W^{f}_j` if `di = 0` and corresponding with `W^{b}_j` if `di = 1` in the equation. Only `ws[0][j]` and `ws[1][j]` where `0 <= j < 1` are `(I, N)` shape as they are multiplied with input variables. All other matrices has `(N, N)` shape.
- **bs** (list of list of *Variable*) – Bias vectors. `bs[i + di]` represnents biases for i-th layer. Note that `di = 0` for forward-RNN and `di = 1` for backward-RNN. Each `bs[i + di]` is a list containing two vectors. `bs[i + di][j]` is corresponding with `b^{f}_j` if `di = 0` and corresponding with `b^{b}_j` if `di = 1` in the equation. Shape of each matrix is `(N,)` where `N` is dimension of hidden units.
- **xs** (list of *Variable*) – A list of *Variable* holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is mini-batch size for time `t`, and `I` is size of input units. Note that this function supports variable length sequences. When sequneces has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. *transpose_sequence()* transpose a list of *Variable()* holding sequence. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.
- **activation** (*str*) – Activation function name. Please select `tanh` or `relu`.

Returns

This function returns a tuple containing three elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[t]` holds hidden states of the last layer corresponding to an input `xs[t]`. Its shape is `(B_t, N)` where `B_t` is mini-batch size for time `t`, and `N` is size of hidden units. Note that `B_t` is the same value as `xs[t]`.

Return type tuple

chainer.functions.n_step_gru

`chainer.functions.n_step_gru(n_layers, dropout_ratio, hx, ws, bs, xs)`

Stacked Uni-directional Gated Recurrent Unit function.

This function calculates stacked Uni-directional GRU with sequences. This function gets an initial hidden state h_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t for each time t from input x_t.

$$
\begin{aligned}
r_t &= \sigma(W_0 x_t + W_3 h_{t-1} + b_0 + b_3) \\
z_t &= \sigma(W_1 x_t + W_4 h_{t-1} + b_1 + b_4) \\
h'_t &= \tanh(W_2 x_t + b_2 + r_t \cdot (W_5 h_{t-1} + b_5)) \\
h_t &= (1 - z_t) \cdot h'_t + z_t \cdot h_{t-1}
\end{aligned}
$$

As the function accepts a sequence, it calculates h_t for all t with one call. Six weight matrices and six bias vectors are required for each layers. So, when S layers exists, you need to prepare $6S$ weight matrices and $6S$ bias vectors.

If the number of layers `n_layers` is greather than 1, input of `k`-th layer is hidden state `h_t` of `k-1`-th layer. Note that all input variables except first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – Number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (`Variable`) – Variable holding stacked hidden states. Its shape is `(S, B, N)` where `S` is number of layers and is equal to `n_layers`, `B` is mini-batch size, and `N` is dimension of hidden units.
- **ws** (list of list of `Variable`) – Weight matrices. `ws[i]` represents weights for i-th layer. Each `ws[i]` is a list containing six matrices. `ws[i][j]` is corresponding with `W_j` in the equation. Only `ws[0][j]` where `0 <= j < 3` is `(I, N)` shape as they are multiplied with input variables. All other matrices has `(N, N)` shape.
- **bs** (list of list of `Variable`) – Bias vectors. `bs[i]` represnents biases for i-th layer. Each `bs[i]` is a list containing six vectors. `bs[i][j]` is corresponding with `b_j` in the equation. Shape of each matrix is `(N,)` where `N` is dimension of hidden units.
- **xs** (list of `Variable`) – A list of `Variable` holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is mini-batch size for time `t`, and `I` is size of input units. Note that this function supports variable length sequences. When sequneces has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. `transpose_sequence()` transpose a list of `Variable()` holding sequence. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of `Variable` . Each element `ys[t]` holds hidden states of the last layer corresponding to an input `xs[t]`. Its shape is `(B_t, N)` where `B_t` is mini-batch size for time `t`, and `N` is size of hidden units. Note that `B_t` is the same value as `xs[t]`.

Return type tuple

chainer.functions.n_step_lstm

`chainer.functions.n_step_lstm(n_layers, dropout_ratio, hx, cx, ws, bs, xs)`

Stacked Uni-directional Long Short-Term Memory function.

This function calculates stacked Uni-directional LSTM with sequences. This function gets an initial hidden state h_0, an initial cell state c_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t and c_t for each time t from input x_t.

$$
\begin{aligned}
i_t &= \sigma(W_0 x_t + W_4 h_{t-1} + b_0 + b_4) \\
f_t &= \sigma(W_1 x_t + W_5 h_{t-1} + b_1 + b_5) \\
o_t &= \sigma(W_2 x_t + W_6 h_{t-1} + b_2 + b_6) \\
a_t &= \tanh(W_3 x_t + W_7 h_{t-1} + b_3 + b_7) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot a_t \\
h_t &= o_t \cdot \tanh(c_t)
\end{aligned}
$$

As the function accepts a sequence, it calculates h_t for all t with one call. Eight weight matrices and eight bias vectors are required for each layer. So, when S layers exist, you need to prepare $8S$ weight matrices and $8S$ bias vectors.

If the number of layers `n_layers` is greater than 1, the input of the `k`-th layer is the hidden state `h_t` of the `k-1`-th layer. Note that all input variables except the first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – The number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*Variable*) – Variable holding stacked hidden states. Its shape is `(S, B, N)` where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **cx** (*Variable*) – Variable holding stacked cell states. It has the same shape as `hx`.
- **ws** (list of list of *Variable*) – Weight matrices. `ws[i]` represents the weights for the i-th layer. Each `ws[i]` is a list containing eight matrices. `ws[i][j]` corresponds to W_j in the equation. Only `ws[0][j]` where `0 <= j < 4` are `(I, N)`-shaped as they are multiplied with input variables, where `I` is the size of the input and `N` is the dimension of the hidden units. All other matrices are `(N, N)`-shaped.
- **bs** (list of list of *Variable*) – Bias vectors. `bs[i]` represents the biases for the i-th layer. Each `bs[i]` is a list containing eight vectors. `bs[i][j]` corresponds to b_j in the equation. The shape of each matrix is `(N,)` where `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – A list of *Variable* holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is the mini-batch size for time `t`. The sequences must be transposed. *transpose_sequence()* can be used to transpose a list of *Variable*s each representing a sequence. When sequences has different lengths, they must be sorted in descending order of their lengths before transposing. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.

Returns

This function returns a tuple containing three elements, `hy`, `cy` and `ys`.

- `hy` is an updated hidden states whose shape is the same as `hx`.
- `cy` is an updated cell states whose shape is the same as `cx`.

- ys is a list of `Variable` . Each element ys[t] holds hidden states of the last layer corresponding to an input xs[t]. Its shape is (B_t, N) where B_t is the mini-batch size for time t, and N is size of hidden units. Note that B_t is the same value as xs[t].

Return type tuple

Note: The dimension of hidden units is limited to only one size N. If you want to use variable dimension of hidden units, please use `chainer.functions.lstm`.

See also:

`chainer.functions.lstm()`

Example

```
>>> batchs = [3, 2, 1]  # support variable length sequences
>>> in_size, out_size, n_layers = 3, 2, 2
>>> dropout_ratio = 0.0
>>> xs = [np.ones((b, in_size)).astype(np.float32) for b in batchs]
>>> [x.shape for x in xs]
[(3, 3), (2, 3), (1, 3)]
>>> h_shape = (n_layers, batchs[0], out_size)
>>> hx = np.ones(h_shape).astype(np.float32)
>>> cx = np.ones(h_shape).astype(np.float32)
>>> w_in = lambda i, j: in_size if i == 0 and j < 4 else out_size
>>> ws = []
>>> bs = []
>>> for n in range(n_layers):
...     ws.append([np.ones((out_size, w_in(n, i))).astype(np.float32) for i in␣
↪range(8)])
...     bs.append([np.ones((out_size,)).astype(np.float32) for _ in range(8)])
...
>>> ws[0][0].shape  # ws[0][:4].shape are (out_size, in_size)
(2, 3)
>>> ws[1][0].shape  # others are (out_size, out_size)
(2, 2)
>>> bs[0][0].shape
(2,)
>>> hy, cy, ys = F.n_step_lstm(
...     n_layers, dropout_ratio, hx, cx, ws, bs, xs)
>>> hy.shape
(2, 3, 2)
>>> cy.shape
(2, 3, 2)
>>> [y.shape for y in ys]
[(3, 2), (2, 2), (1, 2)]
```

chainer.functions.n_step_rnn

chainer.functions.**n_step_rnn**(*n_layers*, *dropout_ratio*, *hx*, *ws*, *bs*, *xs*, *activation='tanh'*)

Stacked Uni-directional RNN function for sequence inputs.

This function calculates stacked Uni-directional RNN with sequences. This function gets an initial hidden state h_0, an initial cell state c_0, an input sequence x, weight matrices W, and bias vectors b. This function calculates hidden states h_t and c_t for each time t from input x_t.

$$h_t = f(W_0 x_t + W_1 h_{t-1} + b_0 + b_1)$$

where f is an activation function.

Weight matrices W contains two matrices W_0 and W_1. W_0 is a parameter for an input sequence. W_1 is a parameter for a hidden state. Bias matrices b contains two matrices b_0 and b_1. b_0 is a parameter for an input sequence. b_1 is a parameter for a hidden state.

As the function accepts a sequence, it calculates h_t for all t with one call. Two weight matrices and two bias vectors are required for each layer. So, when S layers exist, you need to prepare $2S$ weight matrices and $2S$ bias vectors.

If the number of layers `n_layers` is greather than 1, input of `k`-th layer is hidden state `h_t` of `k-1`-th layer. Note that all input variables except first layer may have different shape from the first layer.

Parameters

- **n_layers** (*int*) – Number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*Variable*) – Variable holding stacked hidden states. Its shape is `(S, B, N)` where `S` is number of layers and is equal to `n_layers`, `B` is mini-batch size, and `N` is dimension of hidden units.
- **ws** (list of list of *Variable*) – Weight matrices. `ws[i]` represents weights for i-th layer. Each `ws[i]` is a list containing two matrices. `ws[i][j]` is corresponding with `W_j` in the equation. Only `ws[0][j]` where `0 <= j < 1` is `(I, N)` shape as they are multiplied with input variables. All other matrices has `(N, N)` shape.
- **bs** (list of list of *Variable*) – Bias vectors. `bs[i]` represnents biases for i-th layer. Each `bs[i]` is a list containing two vectors. `bs[i][j]` is corresponding with `b_j` in the equation. Shape of each matrix is `(N,)` where `N` is dimension of hidden units.
- **xs** (list of *Variable*) – A list of *Variable* holding input values. Each element `xs[t]` holds input value for time `t`. Its shape is `(B_t, I)`, where `B_t` is mini-batch size for time `t`, and `I` is size of input units. Note that this function supports variable length sequences. When sequneces has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. `transpose_sequence()` transpose a list of `Variable()` holding sequence. So `xs` needs to satisfy `xs[t].shape[0] >= xs[t + 1].shape[0]`.
- **activation** (*str*) – Activation function name. Please select `tanh` or `relu`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[t]` holds hidden states of the last layer corresponding to an input `xs[t]`. Its shape is `(B_t, N)` where `B_t` is mini-batch size for time `t`, and `N` is size of hidden units. Note that `B_t` is the same value as `xs[t]`.

Return type tuple

chainer.functions.shift

`chainer.functions.shift(x, ksize=3, dilate=1)`

Shift function.

See: Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable of shape (n, c, h, w).
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **dilate** (*int or pair of ints*) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.

Returns Output variable of same shape as `x`.

Return type *Variable*

4.2.5 Evaluation functions

`chainer.functions.accuracy`	Computes multiclass classification accuracy of the minibatch.
`chainer.functions.binary_accuracy`	Computes binary classification accuracy of the minibatch.
`chainer.functions.classification_summary`	Calculates Precision, Recall, F beta Score, and support.
`chainer.functions.f1_score`	
`chainer.functions.precision`	
`chainer.functions.r2_score`	Computes R^2(coefficient of determination) regression score function.
`chainer.functions.recall`	

chainer.functions.accuracy

`chainer.functions.accuracy(y, t, ignore_label=None)`

Computes multiclass classification accuracy of the minibatch.

Parameters

- **y** (*Variable* or *N-dimensional array*) – Array whose (i, j, k, ...)-th element indicates the score of the class j at the (i, k, ...)-th sample. The prediction label $\hat{t}$ is calculated by the formula $\hat{t}(i, k, ...) = \operatorname{argmax}_j y(i, j, k, ...)$.
- **t** (*Variable* or *N-dimensional array*) – Array of ground truth labels.
- **ignore_label** (*int or None*) – Skip calculating accuracy if the true label is `ignore_label`.

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*

Note: This function is non-differentiable.

Example

We show the most common case, when `y` is the two dimensional array.

```
>>> y = np.array([[0.1, 0.7, 0.2], # prediction label is 1
...               [8.0, 1.0, 2.0], # prediction label is 0
...               [-8.0, 1.0, 2.0], # prediction label is 2
...               [-8.0, -1.0, -2.0]]) # prediction label is 1
>>> t = np.array([1, 0, 2, 1], np.int32)
>>> F.accuracy(y, t).array # 100% accuracy because all samples are correct
array(1.)
>>> t = np.array([1, 0, 0, 0], np.int32)
>>> F.accuracy(y, t).array # 50% accuracy because 1st and 2nd samples are correct.
array(0.5)
>>> F.accuracy(y, t, ignore_label=0).array # 100% accuracy because of ignoring
↪the 2nd, 3rd and 4th samples.
array(1.)
```

chainer.functions.binary_accuracy

`chainer.functions.binary_accuracy(y, t)`

Computes binary classification accuracy of the minibatch.

Parameters

- **y** (*Variable* or *N-dimensional array*) – Array whose i-th element indicates the score of positive at the i-th sample. The prediction label $\hat{t}[i]$ is `1` if `y[i] >= 0`, otherwise `0`.
- **t** (*Variable* or *N-dimensional array*) – Array holding a signed integer vector of ground truth labels. If `t[i] == 1`, it indicates that i-th sample is positive. If `t[i] == 0`, it indicates that i-th sample is negative. If `t[i] == -1`, corresponding `y[i]` is ignored. Accuracy is zero if all ground truth labels are `-1`.

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*

Note: This function is non-differentiable.

Example

We show the most common case, when `y` is the two dimensional array.

```
>>> y = np.array([[-2.0, 0.0], # prediction labels are [0, 1]
...               [3.0, -5.0]]) # prediction labels are [1, 0]
>>> t = np.array([[0, 1],
...              [1, 0]], np.int32)
>>> F.binary_accuracy(y, t).array # 100% accuracy because all samples are correct.
array(1.)
>>> t = np.array([[0, 0],
...              [1, 1]], np.int32)
>>> F.binary_accuracy(y, t).array # 50% accuracy because y[0][0] and y[1][0] are
↪correct.
array(0.5)
>>> t = np.array([[0, -1],
...              [1, -1]], np.int32)
```

(continues on next page)

(continued from previous page)

```
>>> F.binary_accuracy(y, t).array # 100% accuracy because of ignoring y[0][1] and
↪y[1][1].
array(1.)
```

chainer.functions.classification_summary

chainer.functions.**classification_summary**(*y*, *t*, *label_num=None*, *beta=1.0*, *ignore_label=-1*)

Calculates Precision, Recall, F beta Score, and support.

This function calculates the following quantities for each class.

- Precision: $\frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fp}}$
- Recall: $\frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}}$
- F beta Score: The weighted harmonic average of Precision and Recall.
- Support: The number of instances of each ground truth label.

Here, `tp`, `fp`, `tn`, and `fn` stand for the number of true positives, false positives, true negatives, and false negatives, respectively.

`label_num` specifies the number of classes, that is, each value in `t` must be an integer in the range of `[0, label_num)`. If `label_num` is `None`, this function regards `label_num` as a maximum of in `t` plus one.

`ignore_label` determines which instances should be ignored. Specifically, instances with the given label are not taken into account for calculating the above quantities. By default, it is set to -1 so that all instances are taken into consideration, as labels are supposed to be non-negative integers. Setting `ignore_label` to a non-negative integer less than `label_num` is illegal and yields undefined behavior. In the current implementation, it arises `RuntimeWarning` and `ignore_label`-th entries in output arrays do not contain correct quantities.

Parameters

- **y** (*Variable* or *N-dimensional array*) – Variable holding a vector of scores.
- **t** (*Variable* or *N-dimensional array*) – Variable holding a vector of ground truth labels.
- **label_num** (*int*) – The number of classes.
- **beta** (*float*) – The parameter which determines the weight of precision in the F-beta score.
- **ignore_label** (*int*) – Instances with this label are ignored.

Returns 4-tuple of ~chainer.Variable of size `(label_num,)`. Each element represents precision, recall, F beta score, and support of this minibatch.

chainer.functions.f1_score

chainer.functions.**f1_score**(*y*, *t*, *label_num=None*, *ignore_label=-1*)

chainer.functions.precision

chainer.functions.**precision**(*y*, *t*, *label_num=None*, *ignore_label=-1*)

chainer.functions.r2_score

`chainer.functions.r2_score(pred, true, sample_weight=None, multioutput='uniform_average')`

Computes R^2(coefficient of determination) regression score function.

Parameters

- **pred** (*Variable* or *N-dimensional array*) – Variable holding a vector, matrix or tensor of estimated target values.
- **true** (*Variable* or *N-dimensional array*) – Variable holding a vector, matrix or tensor of correct target values.
- **sample_weight** – This argument is for compatibility with scikit-learn's implementation of r2_score. Current implementation admits None only.
- **multioutput** (*string*) – ['uniform_average', 'raw_values']. if 'uniform_average', this function returns an average of R^2 score of multiple output. If 'raw_average', this function return a set of R^2 score of multiple output.

Returns A Variable holding a scalar array of the R^2 score if 'multioutput' is 'uniform_average' or a vector of R^2 scores if 'multioutput' is 'raw_values'.

Return type *Variable*

Note: This function is non-differentiable.

chainer.functions.recall

`chainer.functions.recall(y, t, label_num=None, ignore_label=-1)`

4.2.6 Loss functions

chainer.functions.absolute_error	Element-wise absolute error function.
chainer.functions.bernoulli_nll	Computes the negative log-likelihood of a Bernoulli distribution.
chainer.functions.black_out	BlackOut loss function.
chainer.functions.connectionist_temporal_classification	Connectionist Temporal Classification loss function.
chainer.functions.contrastive	Computes contrastive loss.
chainer.functions.crf1d	Calculates negative log-likelihood of linear-chain CRF.
chainer.functions.argmax_crf1d	Computes a state that maximizes a joint probability of the given CRF.
chainer.functions.cross_covariance	Computes the sum-squared cross-covariance penalty between `y` and `z`
chainer.functions.decov	Computes the DeCov loss of `h`
chainer.functions.discriminative_margin_based_clustering_loss	Discriminative margin-based clustering loss function
chainer.functions.gaussian_kl_divergence	Computes the KL-divergence of Gaussian variables from the standard one.
chainer.functions.gaussian_nll	Computes the negative log-likelihood of a Gaussian distribution.

Continued on next page

Table 7 – continued from previous page

chainer.functions.hinge	Computes the hinge loss for a one-of-many classification task.
chainer.functions.huber_loss	Computes the Huber loss.
chainer.functions.mean_absolute_error	Mean absolute error function.
chainer.functions.mean_squared_error	Mean squared error function.
chainer.functions.negative_sampling	Negative sampling loss function.
chainer.functions.sigmoid_cross_entropy	Computes cross entropy loss for pre-sigmoid activations.
chainer.functions.softmax_cross_entropy	Computes cross entropy loss for pre-softmax activations.
chainer.functions.squared_error	Squared error function.
chainer.functions.triplet	Computes triplet loss.

chainer.functions.absolute_error

`chainer.functions.absolute_error(x0, x1)`

Element-wise absolute error function.

Computes the element-wise absolute error L between two inputs x_0 and x_1 defined as follows.

$$L = |x_0 - x_1|$$

Parameters

- **x0** (`Variable` or *N-dimensional array*) – First input variable.
- **x1** (`Variable` or *N-dimensional array*) – Second input variable.

Returns An array representing the element-wise absolute error between the two inputs.

Return type *Variable*

chainer.functions.bernoulli_nll

`chainer.functions.bernoulli_nll(x, y, reduce='sum')`

Computes the negative log-likelihood of a Bernoulli distribution.

This function calculates the negative log-likelihood of a Bernoulli distribution.

$$-\log B(x;p) = -\sum_i \{x_i \log(p_i) + (1 - x_i)\log(1 - p_i)\},$$

where $p = \sigma(y)$, $\sigma(\cdot)$ is a sigmoid function, and $B(x;p)$ is a Bernoulli distribution.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'sum'` or `'mean'`, loss values are summed up or averaged respectively.

Note: As this function uses a sigmoid function, you can pass a result of fully-connected layer (that means `Linear`) to this function directly.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.

- **y** (*Variable* or *N-dimensional array*) – A variable representing the parameter of Bernoulli distribution.
- **reduce** (*str*) – Reduction option. Its value must be either `'sum'`, `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable representing the negative log-likelihood. If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'sum'` or `'mean'`, the output variable holds a scalar value.

Return type *Variable*

chainer.functions.black_out

`chainer.functions.black_out(x, t, W, samples, reduce='mean')`

BlackOut loss function.

BlackOut loss function is defined as

$$-\log(p(t)) - \sum_{s \in S} \log(1 - p(s)),$$

where t is the correct label, S is a set of negative examples and $p(\cdot)$ is likelihood of a given label. And, p is defined as

$$p(y) = \frac{\exp(W_y^\top x)}{\sum_{s \in samples} \exp(W_s^\top x)}.$$

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the no loss values. If it is `'mean'`, this function takes a mean of loss values.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Batch of input vectors. Its shape should be (N, D).
- **t** (*Variable* or *N-dimensional array*) – Vector of ground truth labels. Its shape should be $(N,)$. Each elements v should satisfy $0 \geq v \geq V$ or -1 where V is the number of label types.
- **W** (*Variable* or *N-dimensional array*) – Weight matrix. Its shape should be (V, D)
- **samples** (`Variable`) – Negative samples. Its shape should be (N, S) where S is the number of negative samples.
- **reduce** (*str*) – Reduction option. Its value must be either `'no'` or `'mean'`. Otherwise, `ValueError` is raised.

Returns A variable object holding loss value(s). If `reduce` is `'no'`, the output variable holds an array whose shape is $(N,)$. If it is `'mean'`, it holds a scalar.

Return type *Variable*

See: BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies

See also:

BlackOut to manage the model parameter `W`.

chainer.functions.connectionist_temporal_classification

`chainer.functions.connectionist_temporal_classification(x, t, blank_symbol, input_length=None, label_length=None, reduce='mean')`

Connectionist Temporal Classification loss function.

Connectionist Temporal Classification(CTC) [Graves2006] is a loss function of sequence labeling where the alignment between the inputs and target is unknown. See also [Graves2012]

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the samplewise loss values. If it is `'mean'`, it takes the mean of loss values.

Parameters

- **x** (list or tuple of `Variable`) – A list of unnormalized probabilities for labels. Each element of `x`, `x[i]` is a `Variable` object, which has shape `(B, V)`, where `B` is the batch size and `V` is the number of labels. The softmax of `x[i]` represents the probabilities of the labels at time `i`.
- **t** (`Variable` or *N-dimensional array*) – A matrix including expected label sequences. Its shape is `(B, M)`, where `B` is the batch size and `M` is the maximum length of the label sequences. All elements in `t` must be less than `V`, the number of labels.
- **blank_symbol** (`int`) – Index of blank_symbol. This value must be non-negative.
- **input_length** (`Variable` or *N-dimensional array*) – Length of sequence for each of mini batch `x` (optional). Its shape must be `(B,)`. If the `input_length` is omitted or `None`, it assumes that all of `x` is valid input.
- **label_length** (`Variable` or *N-dimensional array*) – Length of sequence for each of mini batch `t` (optional). Its shape must be `(B,)`. If the `label_length` is omitted or `None`, it assumes that all of `t` is valid input.
- **reduce** (`str`) – Reduction option. Its value must be either `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding a scalar value of the CTC loss. If `reduce` is `'no'`, the output variable holds array whose shape is *(B,)* where *B* is the number of samples. If it is `'mean'`, it holds a scalar.

Return type *Variable*

Note: You need to input `x` without applying to activation functions(e.g. softmax function), because this function applies softmax functions to `x` before calculating CTC loss to avoid numerical limitations. You also need to apply softmax function to forwarded values before you decode it.

Note: This function is differentiable only by `x`.

Note: This function supports (batch, sequence, 1-dimensional input)-data.

chainer.functions.contrastive

`chainer.functions.contrastive(x0, x1, y, margin=1, reduce='mean')`

Computes contrastive loss.

It takes a pair of samples and a label as inputs. The label is 1 when those samples are similar, or 0 when they are dissimilar.

Let N and K denote mini-batch size and the dimension of input variables, respectively. The shape of both input variables `x0` and `x1` should be `(N, K)`. The loss value of the n-th sample pair L_n is

$$L_n = \frac{1}{2} \left(y_n d_n^2 + (1 - y_n) \max(\text{margin} - d_n, 0)^2 \right)$$

where $d_n = \|\mathbf{x_0}_n - \mathbf{x_1}_n\|_2$, $\mathbf{x_0}_n$ and $\mathbf{x_1}_n$ are n-th K-dimensional vectors of `x0` and `x1`.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'mean'`, this function takes a mean of loss values.

Parameters

- **x0** (`Variable` or *N-dimensional array*) – The first input variable. The shape should be (N, K), where N denotes the mini-batch size, and K denotes the dimension of `x0`.
- **x1** (`Variable` or *N-dimensional array*) – The second input variable. The shape should be the same as `x0`.
- **y** (`Variable` or *N-dimensional array*) – Labels. All values should be 0 or 1. The shape should be `(N,)`, where N denotes the mini-batch size.
- **margin** (*float*) – A parameter for contrastive loss. It should be positive value.
- **reduce** (*str*) – Reduction option. Its value must be either `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding the loss value(s) calculated by the above equation. If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'mean'`, the output variable holds a scalar value.

Return type *Variable*

Note: This cost can be used to train siamese networks. See Learning a Similarity Metric Discriminatively, with Application to Face Verification for details.

Example

```
>>> x0 = np.array([[-2.0, 3.0, 0.5], [5.0, 2.0, -0.5]]).astype(np.float32)
>>> x1 = np.array([[-1.0, 3.0, 1.0], [3.5, 0.5, -2.0]]).astype(np.float32)
>>> y = np.array([1, 0]).astype(np.int32)
>>> F.contrastive(x0, x1, y)
variable(0.3125)
>>> F.contrastive(x0, x1, y, margin=3.0)  # harder penalty
variable(0.3528857)
>>> z = F.contrastive(x0, x1, y, reduce='no')
>>> z.shape
(2,)
>>> z.array
array([0.625, 0.   ], dtype=float32)
```

chainer.functions.crf1d

`chainer.functions.crf1d(cost, xs, ys, reduce='mean')`

Calculates negative log-likelihood of linear-chain CRF.

It takes a transition cost matrix, a sequence of costs, and a sequence of labels. Let c_{st} be a transition cost from a label s to a label t, x_{it} be a cost of a label t at position i, and y_i be an expected label at position i. The negative log-likelihood of linear-chain CRF is defined as

$$L = -\left(\sum_{i=1}^{l} x_{iy_i} + \sum_{i=1}^{l-1} c_{y_i y_{i+1}} - \log(Z) \right),$$

where l is the length of the input sequence and Z is the normalizing constant called partition function.

Note: When you want to calculate the negative log-likelihood of sequences which have different lengths, sort the sequences in descending order of lengths and transpose the sequences. For example, you have three input sequences:

```
>>> a1 = a2 = a3 = a4 = np.random.uniform(-1, 1, 3).astype(np.float32)
>>> b1 = b2 = b3 = np.random.uniform(-1, 1, 3).astype(np.float32)
>>> c1 = c2 = np.random.uniform(-1, 1, 3).astype(np.float32)
```

```
>>> a = [a1, a2, a3, a4]
>>> b = [b1, b2, b3]
>>> c = [c1, c2]
```

where `a1` and all other variables are arrays with `(K,)` shape. Make a transpose of the sequences:

```
>>> x1 = np.stack([a1, b1, c1])
>>> x2 = np.stack([a2, b2, c2])
>>> x3 = np.stack([a3, b3])
>>> x4 = np.stack([a4])
```

and make a list of the arrays:

```
>>> xs = [x1, x2, x3, x4]
```

You need to make label sequences in the same fashion. And then, call the function:

```
>>> cost = chainer.Variable(
...     np.random.uniform(-1, 1, (3, 3)).astype(np.float32))
>>> ys = [np.zeros(x.shape[0:1], dtype=np.int32) for x in xs]
>>> loss = F.crf1d(cost, xs, ys)
```

It calculates mean of the negative log-likelihood of the three sequences.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'mean'`, it holds mean of the loss values.

Parameters

- **cost** (`Variable` or *N-dimensional array*) – A $K \times K$ matrix which holds transition cost between two labels, where K is the number of labels.
- **xs** (`list of Variable`) – Input vector for each label. `len(xs)` denotes the length of the sequence, and each `Variable` holds a $B \times K$ matrix, where B is mini-batch size, K

is the number of labels. Note that Bs in all the variables are not necessary the same, i.e., it accepts the input sequences with different lengths.

- **ys** (*list of Variable*) – Expected output labels. It needs to have the same length as `xs`. Each `Variable` holds a B integer vector. When `x` in `xs` has the different B, correspoding `y` has the same B. In other words, `ys` must satisfy `ys[i].shape == xs[i].shape[0:1]` for all `i`.
- **reduce** (*str*) – Reduction option. Its value must be either `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding the average negative log-likelihood of the input sequences.

Return type *Variable*

Note: See detail in the original paper: Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data.

chainer.functions.argmax_crf1d

`chainer.functions.argmax_crf1d(cost, xs)`

Computes a state that maximizes a joint probability of the given CRF.

Parameters

- **cost** (`Variable` or *N-dimensional array*) – A $K \times K$ matrix which holds transition cost between two labels, where K is the number of labels.
- **xs** (*list of Variable*) – Input vector for each label. `len(xs)` denotes the length of the sequence, and each `Variable` holds a $B \times K$ matrix, where B is mini-batch size, K is the number of labels. Note that Bs in all the variables are not necessary the same, i.e., it accepts the input sequences with different lengths.

Returns A tuple of `Variable` object `s` and a `list` `ps`. The shape of `s` is `(B,)`, where `B` is the mini-batch size. i-th element of `s`, `s[i]`, represents log-likelihood of i-th data. `ps` is a list of *N-dimensional array*, and denotes the state that maximizes the point probability. `len(ps)` is equal to `len(xs)`, and shape of each `ps[i]` is the mini-batch size of the corresponding `xs[i]`. That means, `ps[i].shape == xs[i].shape[0:1]`.

Return type tuple

chainer.functions.cross_covariance

`chainer.functions.cross_covariance(y, z, reduce='half_squared_sum')`

Computes the sum-squared cross-covariance penalty between `y` and `z`

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the covariant matrix that has as many rows (resp. columns) as the dimension of `y` (resp.z). If it is `'half_squared_sum'`, it holds the half of the Frobenius norm (i.e. L2 norm of a matrix flattened to a vector) of the covariianct matrix.

Parameters

- **y** (`Variable` or *N-dimensional array*) – Variable holding a matrix where the first dimension corresponds to the batches.

- **z** (*Variable* or *N-dimensional array*) – Variable holding a matrix where the first dimension corresponds to the batches.
- **reduce** (*str*) – Reduction option. Its value must be either `'half_squared_sum'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding the cross covariance loss. If `reduce` is `'no'`, the output variable holds 2-dimensional array matrix of shape `(M, N)` where `M` (resp. `N`) is the number of columns of `y` (resp. `z`). If it is `'half_squared_sum'`, the output variable holds a scalar value.

Return type *Variable*

Note: This cost can be used to disentangle variables. See https://arxiv.org/abs/1412.6583v3 for details.

---

chainer.functions.decov

`chainer.functions.decov(h, reduce='half_squared_sum')`

Computes the DeCov loss of `h`

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds a matrix whose size is same as the number of columns of `y`. If it is `'half_squared_sum'`, it holds the half of the squared Frobenius norm (i.e. squared of the L2 norm of a matrix flattened to a vector) of the matrix.

Parameters

- **h** (*Variable* or *N-dimensional array*) – Variable holding a matrix where the first dimension corresponds to the batches.
- **recude** (*str*) – Reduction option. Its value must be either `'half_squared_sum'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding a scalar of the DeCov loss. If `reduce` is `'no'`, the output variable holds 2-dimensional array matrix of shape `(N, N)` where `N` is the number of columns of `y`. If it is `'half_squared_sum'`, the output variable holds a scalar value.

Return type *Variable*

Note: See https://arxiv.org/abs/1511.06068 for details.

---

chainer.functions.discriminative_margin_based_clustering_loss

`chainer.functions.discriminative_margin_based_clustering_loss(embeddings, labels, delta_v, delta_d, max_embedding_dim, norm=1, alpha=1.0, beta=1.0, gamma=0.001)`

Discriminative margin-based clustering loss function

This is the implementation of the following paper: https://arxiv.org/abs/1708.02551 This method is a semi-supervised solution to instance segmentation. It calculates pixel embeddings, and calculates three different terms based on those embeddings and applies them as loss. The main idea is that the pixel embeddings for same instances have to be closer to each other (pull force), for different instances, they have to be further away (push force). The loss also brings a weak regularization term to prevent overfitting. This loss function calculates the following three parameters:

Variance Loss Loss to penalize distances between pixels which are belonging to the same instance. (Pull force)

Distance loss Loss to penalize distances between the centers of instances. (Push force)

Regularization loss Small regularization loss to penalize weights against overfitting.

Parameters

- **embeddings** (*Variable* or *N-dimensional array*) – predicted embedding vectors (batch size, max embedding dimensions, height, width)
- **labels** (*N-dimensional array*) – instance segmentation ground truth each unique value has to be denoting one instance (batch size, height, width)
- **delta_v** (*float*) – Minimum distance to start penalizing variance
- **delta_d** (*float*) – Maximum distance to stop penalizing distance
- **max_embedding_dim** (*int*) – Maximum number of embedding dimensions
- **norm** (*int*) – Norm to calculate pixels and cluster center distances
- **alpha** (*float*) – Weight for variance loss
- **beta** (*float*) – Weight for distance loss
- **gamma** (*float*) – Weight for regularization loss

Returns

- *Variance loss*: Variance loss multiplied by `alpha`
- *Distance loss*: Distance loss multiplied by `beta`
- *Regularization loss*: Regularization loss multiplied by `gamma`

Return type `tuple` of *chainer.Variable*

chainer.functions.gaussian_kl_divergence

`chainer.functions.`**gaussian_kl_divergence**(*mean*, *ln_var*, *reduce='sum'*)

Computes the KL-divergence of Gaussian variables from the standard one.

Given two variable `mean` representing μ and `ln_var` representing $\log(\sigma^2)$, this function calculates the KL-divergence in elementwise manner between the given multi-dimensional Gaussian $N(\mu, S)$ and the standard Gaussian $N(0, I)$

$$D_{\mathbf{KL}}(N(\mu, S) \| N(0, I)),$$

where S is a diagonal matrix such that $S_{ii} = \sigma_i^2$ and I is an identity matrix.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'sum'` or `'mean'`, loss values are summed up or averaged respectively.

Parameters

- **mean** (*Variable* or *N-dimensional array*) – A variable representing mean of given gaussian distribution, μ.
- **ln_var** (*Variable* or *N-dimensional array*) – A variable representing logarithm of variance of given gaussian distribution, $\log(\sigma^2)$.

- **reduce** (*str*) – Reduction option. Its value must be either 'sum', 'mean' or 'no'. Otherwise, ValueError is raised.

Returns A variable representing KL-divergence between given gaussian distribution and the standard gaussian. If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum' or 'mean', the output variable holds a scalar value.

Return type *Variable*

chainer.functions.gaussian_nll

chainer.functions.**gaussian_nll**(*x*, *mean*, *ln_var*, *reduce='sum'*)

Computes the negative log-likelihood of a Gaussian distribution.

Given two variable mean representing μ and ln_var representing $\log(\sigma^2)$, this function computes in elementwise manner the negative log-likelihood of x on a Gaussian distribution $N(\mu, S)$,

$$-\log N(x;\mu,\sigma^2) = \log\left(\sqrt{(2\pi)^D|S|}\right) + \frac{1}{2}(x-\mu)^\top S^{-1}(x-\mu),$$

where D is a dimension of x and S is a diagonal matrix where $S_{ii} = \sigma_i^2$.

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'sum' or 'mean', loss values are summed up or averaged respectively.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **mean** (*Variable* or *N-dimensional array*) – A variable representing mean of a Gaussian distribution, μ.
- **ln_var** (*Variable* or *N-dimensional array*) – A variable representing logarithm of variance of a Gaussian distribution, $\log(\sigma^2)$.
- **reduce** (*str*) – Reduction option. Its value must be either 'sum', 'mean' or 'no'. Otherwise, ValueError is raised.

Returns A variable representing the negative log-likelihood. If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum' or 'mean', the output variable holds a scalar value.

Return type *Variable*

chainer.functions.hinge

chainer.functions.**hinge**(*x*, *t*, *norm='L1'*, *reduce='mean'*)

Computes the hinge loss for a one-of-many classification task.

$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left[\max(0, 1-\delta\{t_n = k\}x_{nk})\right]^p$$

where N denotes the batch size and K is the number of classes of interest,

$$\delta\{\text{condition}\} = \left\{\begin{array}{cc} 1 & \text{if condition is true} \\ -1 & \text{otherwise,} \end{array}\right.$$

and

$$p = \begin{cases} 1 & \text{if norm} = \text{L1} \\ 2 & \text{if norm} = \text{L2.} \end{cases}$$

Let the hinge loss function $l(x, \delta)$ be $[\max(0, 1 - \delta x)]^p$. When x and δ have the same sign (meaning x predicts the proper score for classification) and $|x| \geq 1$, the hinge loss $l(x, \delta) = 0$, but when they have opposite sign, $l(x, \delta)$ increases linearly with x.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'mean'`, it takes the mean of loss values.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. The shape of `x` should be (N, K).
- **t** (*Variable* or *N-dimensional array*) – The N-dimensional label vector with values $t_n \in \{0, 1, 2, \ldots, K-1\}$. The shape of `t` should be $(N,)$.
- **norm** (*string*) – Specifies norm type. Either `'L1'` or `'L2'` is acceptable.
- **reduce** (*str*) – Reduction option. Its value must be either `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable object holding a scalar array of the hinge loss L. If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'mean'`, the output variable holds a scalar value.

Return type *Variable*

Example

In this case, the batch size `N` is 2 and the number of classes `K` is 3.

```
>>> x = np.array([[-2.0, 3.0, 0.5],
...               [5.0, 2.0, -0.5]]).astype(np.float32)
>>> x
array([[-2. ,  3. ,  0.5],
       [ 5. ,  2. , -0.5]], dtype=float32)
>>> t = np.array([1, 0]).astype(np.int32)
>>> t
array([1, 0], dtype=int32)
>>> F.hinge(x, t)
variable(2.5)
>>> F.hinge(x, t, reduce='no')
variable([[0. , 0. , 1.5],
          [0. , 3. , 0.5]])
>>> F.hinge(x, t, norm='L2')
variable(5.75)
```

chainer.functions.huber_loss

`chainer.functions.huber_loss`(*x*, *t*, *delta*, *reduce='sum_along_second_axis'*)

Computes the Huber loss.

The Huber loss is similar to the *mean_squared_error()* but is less sensitive to outliers in the data. It is defined as

$$L_{\delta}(a) = \left\{ \begin{array}{cc} \frac{1}{2}a^2 & \text{if |a|} \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise,} \end{array} \right.$$

where $a = x - t$ is the difference between the input x and the target t.

The loss is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'sum_along_second_axis'`, loss values are summed up along the second axis (i.e. `axis=1`).

See: Huber loss - Wikipedia.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. The shape of `x` should be $(N, K, \ldots)$ if `reduce='sum_along_second_axis'`.
- **t** (*Variable* or *N-dimensional array*) – Target variable for regression. The shape of `t` should be $(N, K, \ldots)$ if `reduce='sum_along_second_axis'`.
- **delta** (*float*) – Constant variable for Huber loss function as used in definition.
- **reduce** (*str*) – Reduction option. Its value must be either `'sum_along_second_axis'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable object holding a scalar array of the Huber loss L_{δ}. If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'sum_along_second_axis'`, the shape of the array is same as the input variables, except the second axis is removed.

Return type *Variable*

Example

Example without reduction, in which case the output `y` will have the same shape as the inputs `x` and `t`.

```
>>> import numpy as np
>>> from chainer import functions as F
>>> x = np.array([[-2.0, 3.0, 0.5], [5.0, 2.0, -0.5]]).astype(np.float32)
>>> x.shape
(2, 3)
>>> t = np.array([[-2.0, 3.0, 0.0], [10.0, 2.0, -0.5]]).astype(np.float32)
>>> t.shape
(2, 3)
>>> y = F.huber_loss(x, t, delta=1.0, reduce='no')
>>> y.shape
(2, 3)
>>> y
variable([[0.   , 0.   , 0.125],
          [4.5  , 0.   , 0.   ]])
```

Example with reduction along the second axis.

```
>>> y = F.huber_loss(x, t, delta=1.0, reduce='sum_along_second_axis')
>>> y.shape
(2,)
>>> y
variable([0.125, 4.5  ])
```

chainer.functions.mean_absolute_error

`chainer.functions.mean_absolute_error(x0, x1)`

Mean absolute error function.

The function computes the mean absolute error between two variables. The mean is taken over the minibatch. Args `x0` and `x1` must have the same dimensions. This function first calculates the absolute value differences between the corresponding elements in x0 and x1, and then returns the mean of those differences.

Parameters

- **x0** (`Variable` or *N-dimensional array*) – Input variable.
- **x1** (`Variable` or *N-dimensional array*) – Input variable.

Returns A variable holding an array representing the mean absolute error of two inputs.

Return type *Variable*

Example

1D array examples:

```
>>> x = np.array([1, 2, 3]).astype(np.float32)
>>> y = np.array([0, 0, 0]).astype(np.float32)
>>> F.mean_absolute_error(x, y)
variable(2.)
>>> x = np.array([1, 2, 3, 4, 5, 6]).astype(np.float32)
>>> y = np.array([7, 8, 9, 10, 11, 12]).astype(np.float32)
>>> F.mean_absolute_error(x, y)
variable(6.)
```

2D array example:

In this example, there are 4 elements, and thus 4 errors >>> x = np.array([[1, 2], [3, 4]]).astype(np.float32) >>> y = np.array([[8, 8], [8, 8]]).astype(np.float32) >>> F.mean_absolute_error(x, y) variable(5.5)

3D array example:

In this example, there are 8 elements, and thus 8 errors >>> x = np.reshape(np.array([1, 2, 3, 4, 5, 6, 7, 8]), (2, 2, 2)) >>> y = np.reshape(np.array([8, 8, 8, 8, 8, 8, 8, 8]), (2, 2, 2)) >>> x = x.astype(np.float32) >>> y = y.astype(np.float32) >>> F.mean_absolute_error(x, y) variable(3.5)

---

chainer.functions.mean_squared_error

`chainer.functions.mean_squared_error(x0, x1)`

Mean squared error function.

The function computes the mean squared error between two variables. The mean is taken over the minibatch. Args `x0` and `x1` must have the same dimensions. Note that the error is not scaled by 1/2.

Parameters

- **x0** (`Variable` or *N-dimensional array*) – Input variable.
- **x1** (`Variable` or *N-dimensional array*) – Input variable.

Returns A variable holding an array representing the mean squared error of two inputs.

Return type

~chainer.Variable

Example

1D array examples:

```
>>> x = np.array([1, 2, 3, 4]).astype(np.float32)
>>> y = np.array([0, 0, 0, 0]).astype(np.float32)
>>> F.mean_squared_error(x, y)
variable(7.5)
>>> x = np.array([1, 2, 3, 4, 5, 6]).astype(np.float32)
>>> y = np.array([7, 8, 9, 10, 11, 12]).astype(np.float32)
>>> F.mean_squared_error(x, y)
variable(36.)
```

2D array example:

In this example, there are 4 elements, and thus 4 errors >>> x = np.array([[1, 2], [3, 4]]).astype(np.float32) >>> y = np.array([[8, 8], [8, 8]]).astype(np.float32) >>> F.mean_squared_error(x, y) variable(31.5)

3D array example:

In this example, there are 8 elements, and thus 8 errors >>> x = np.reshape(np.array([1, 2, 3, 4, 5, 6, 7, 8]), (2, 2, 2)) >>> y = np.reshape(np.array([8, 8, 8, 8, 8, 8, 8, 8]), (2, 2, 2)) >>> x = x.astype(np.float32) >>> y = y.astype(np.float32) >>> F.mean_squared_error(x, y) variable(17.5)

---

chainer.functions.negative_sampling

`chainer.functions.negative_sampling`(*x*, *t*, *W*, *sampler*, *sample_size*, *reduce='sum'*, ***, *return_samples=False*)

Negative sampling loss function.

In natural language processing, especially language modeling, the number of words in a vocabulary can be very large. Therefore, you need to spend a lot of time calculating the gradient of the embedding matrix.

By using the negative sampling trick you only need to calculate the gradient for a few sampled negative examples.

The loss is defined as follows.

$$f(x,p) = -\log\sigma(x^\top w_p) - \ kE_{i\sim P(i)}[\log\sigma(-x^\top w_i)]$$

where $\sigma(\cdot)$ is a sigmoid function, w_i is the weight vector for the word i, and p is a positive example. It is approximated with k examples N sampled from probability $P(i)$.

$$f(x,p) \approx -\log\sigma(x^\top w_p) - \sum_{n\in N}\log\sigma(-x^\top w_n)$$

Each sample of N is drawn from the word distribution $P(w) = \frac{1}{Z}c(w)^\alpha$, where $c(w)$ is the unigram count of the word w, α is a hyper-parameter, and Z is the normalization constant.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Batch of input vectors.

- **t** (*Variable* or *N-dimensional array*) – Vector of ground truth labels.
- **W** (*Variable* or *N-dimensional array*) – Weight matrix.
- **sampler** (*FunctionType*) – Sampling function. It takes a shape and returns an integer array of the shape. Each element of this array is a sample from the word distribution. A *WalkerAlias* object built with the power distribution of word frequency is recommended.
- **sample_size** (*int*) – Number of samples.
- **reduce** (*str*) – Reduction option. Its value must be either `'sum'` or `'no'`. Otherwise, `ValueError` is raised.

- **return_samples** (*bool*) – If `True`, the sample array is also returned. The sample array is a (

Returns

If `return_samples` is `False` (default), the output variable holding the loss value(s) calculated by the above equation is returned. Otherwise, a tuple of the output variable and the sample array is returned.

If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'sum'`, the output variable holds a scalar value.

Return type *Variable* or tuple

See: Distributed Representations of Words and Phrases and their Compositionality

See also:

NegativeSampling to manage the model parameter `W`.

chainer.functions.sigmoid_cross_entropy

`chainer.functions.`**`sigmoid_cross_entropy`**(*x, t, normalize=True, reduce='mean'*)

Computes cross entropy loss for pre-sigmoid activations.

Parameters

- **x** (*Variable* or *N-dimensional array*) – A variable object holding a matrix whose (i, j)-th element indicates the unnormalized log probability of the j-th unit at the i-th example.
- **t** (*Variable* or *N-dimensional array*) – A variable object holding a matrix whose (i, j)-th element indicates a signed integer vector of ground truth labels 0 or 1. If `t[i, j] == -1`, corresponding `x[i, j]` is ignored. Loss is zero if all ground truth labels are `-1`.
- **normalize** (*bool*) – Variable holding a boolean value which determines the normalization constant. If true, this function normalizes the cross entropy loss across all instances. If else, it only normalizes along a batch size.
- **reduce** (*str*) – Variable holding a `str` which determines whether to reduce the shape of the input. If it is `'mean'`, it computes the sum of cross entropy and normalize it according to `normalize` option. If is is `'no'`, this function computes cross entropy for each instance and does not normalize it (`normalize` option is ignored). In this case, the loss value of the ignored instance, which has `-1` as its target value, is set to `0`.

Returns A variable object holding an array of the cross entropy. If `reduce` is `'mean'`, it is a scalar array. If `reduce` is `'no'`, the shape is same as those of `x` and `t`.

Return type *Variable*

Note: This function is differentiable only by `x`.

Example

```
>>> x = np.array([[-2.0, 3.0, 0.5], [5.0, 2.0, -0.5]]).astype(np.float32)
>>> x
array([[-2. ,  3. ,  0.5],
       [ 5. ,  2. , -0.5]], dtype=float32)
>>> t = np.array([[0, 1, 0], [1, 1, -1]]).astype(np.int32)
>>> t
array([[ 0,  1,  0],
       [ 1,  1, -1]], dtype=int32)
>>> F.sigmoid_cross_entropy(x, t)
variable(0.25664714)
>>> F.sigmoid_cross_entropy(x, t, normalize=False)
variable(0.64161783)
>>> y = F.sigmoid_cross_entropy(x, t, reduce='no')
>>> y.shape
(2, 3)
>>> y.array
array([[ 0.126928  ,  0.04858735,  0.974077  ],
       [ 0.00671535,  0.126928  , -0.        ]], dtype=float32)
```

chainer.functions.softmax_cross_entropy

`chainer.functions.`**`softmax_cross_entropy`**(*x*, *t*, *normalize=True*, *cache_score=True*, *class_weight=None*, *ignore_label=-1*, *reduce='mean'*, *enable_double_backprop=False*)

Computes cross entropy loss for pre-softmax activations.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Variable holding a multidimensional array whose element indicates unnormalized log probability: the first axis of the variable represents the number of samples, and the second axis represents the number of classes. While this function computes a usual softmax cross entropy if the number of dimensions is equal to 2, it computes a cross entropy of the replicated softmax if the number of dimensions is greater than 2.
- **t** (*Variable* or *N-dimensional array*) – Variable holding a signed integer vector of ground truth labels. If `t[i] == ignore_label`, corresponding `x[i]` is ignored.
- **normalize** (*bool*) – If `True`, this function normalizes the cross entropy loss across all instances. If `False`, it only normalizes along a batch size.
- **cache_score** (*bool*) – When it is `True`, the function stores result of forward computation to use it on backward computation. It reduces computational cost though consumes more memory. If `enable_double_backprop` option is `True`, this option is forcibly turned off and the function does not cache the intermediate value.
- **class_weight** (*N-dimensional array*) – An array that contains constant weights that will be multiplied with the loss values along with the second dimension. The shape of this array should be `(x.shape[1],)`. If this is not `None`, each class weight `class_weight[i]`

is actually multiplied to `y[:, i]` that is the corresponding log-softmax output of `x` and has the same shape as `x` before calculating the actual loss value.

- **ignore_label** (*int*) – Label value you want to ignore. Its default value is `-1`. See description of the argument *t*.
- **reduce** (*str*) – A string that determines whether to reduce the loss values. If it is `'mean'`, it computes the sum of the individual cross entropy and normalize it according to `normalize` option. If it is `'no'`, this function computes cross entropy for each instance and does not normalize it (`normalize` option is ignored). In this case, the loss value of the ignored instance, which has `ignore_label` as its target value, is set to `0`.
- **enable_double_backprop** (*bool*) – If `True`, this function uses implementation that supports higher order differentiation. If `False`, it uses single-backprop implementation. This function use the single-backprop version because we expect it is faster. So, if you need second or higher derivatives, you need to turn it on explicitly.

Returns A variable holding a scalar array of the cross entropy loss. If `reduce` is `'mean'`, it is a scalar array. If `reduce` is `'no'`, the shape is same as that of `t`.

Return type *Variable*

Note: This function is differentiable only by `x`.

Example

```
>>> x = np.array([[-1, 0, 1, 2], [2, 0, 1, -1]]).astype(np.float32)
>>> x
array([[-1.,  0.,  1.,  2.],
       [ 2.,  0.,  1., -1.]], dtype=float32)
>>> t = np.array([3, 0]).astype(np.int32)
>>> t
array([3, 0], dtype=int32)
>>> y = F.softmax_cross_entropy(x, t)
>>> y
variable(0.44018972)
>>> log_softmax = -F.log_softmax(x)
>>> expected_loss = np.mean([log_softmax[row, column].data for row, column in␣
↪enumerate(t)])
>>> y.array == expected_loss
True
```

chainer.functions.squared_error

`chainer.functions.`**`squared_error`**`(`*x0*, *x1*`)`

Squared error function.

This function computes the squared error between two variables:

$$(x_0 - x_1)^2$$

where operation is done in elementwise manner. Note that the error is not scaled by 1/2:

Parameters

- **x0** (`Variable` or *N-dimensional array*) – Input variable.

- **x1** (*Variable* or *N-dimensional array*) – Input variable.

Returns A variable holding an array representing the squared error of two inputs.

Return type *Variable*

Note: *squared_error()* and *squared_difference()* are identical functions, aside from the different argument names. They are both kept for backward compatibility.

See also:

squared_difference()

Example

```
>>> x1 = np.arange(6).astype(np.float32)
>>> x1
array([0., 1., 2., 3., 4., 5.], dtype=float32)
>>> x2 = np.array([5, 4, 3, 2, 1, 0]).astype(np.float32)
>>> x2
array([5., 4., 3., 2., 1., 0.], dtype=float32)
>>> y = F.squared_error(x1, x2)
>>> y.shape
(6,)
>>> y.array
array([25.,  9.,  1.,  1.,  9., 25.], dtype=float32)
```

See also:

squared_difference()

chainer.functions.triplet

`chainer.functions.triplet`(*anchor*, *positive*, *negative*, *margin=0.2*, *reduce='mean'*)

Computes triplet loss.

It takes a triplet of variables as inputs, a, p and n: anchor, positive example and negative example respectively. The triplet defines a relative similarity between samples. Let N and K denote mini-batch size and the dimension of input variables, respectively. The shape of all input variables should be (N, K).

$$L(a,p,n) = \frac{1}{N}\left(\sum_{i=1}^{N} \max\{d(a_i,p_i) - d(a_i,n_i) + \mathrm{margin}, 0\}\right)$$

where $d(x_i, y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_2^2$.

The output is a variable whose value depends on the value of the option `reduce`. If it is `'no'`, it holds the elementwise loss values. If it is `'mean'`, this function takes a mean of loss values.

Parameters

- **anchor** (*Variable* or *N-dimensional array*) – The anchor example variable. The shape should be (N, K), where N denotes the minibatch size, and K denotes the dimension of the anchor.
- **positive** (*Variable* or *N-dimensional array*) – The positive example variable. The shape should be the same as anchor.

- **negative** (*Variable* or *N-dimensional array*) – The negative example variable. The shape should be the same as anchor.
- **margin** (*float*) – A parameter for triplet loss. It should be a positive value.
- **reduce** (*str*) – Reduction option. Its value must be either `'mean'` or `'no'`. Otherwise, `ValueError` is raised.

Returns A variable holding a scalar that is the loss value calculated by the above equation. If `reduce` is `'no'`, the output variable holds array whose shape is same as one of (hence both of) input variables. If it is `'mean'`, the output variable holds a scalar value.

Return type *Variable*

Note: This cost can be used to train triplet networks. See Learning Fine-grained Image Similarity with Deep Ranking for details.

Example

```
>>> anchor = np.array([[-2.0, 3.0, 0.5], [5.0, 2.0, -0.5]]).astype(np.float32)
>>> pos = np.array([[-2.1, 2.8, 0.5], [4.9, 2.0, -0.4]]).astype(np.float32)
>>> neg = np.array([[-2.1, 2.7, 0.7], [4.9, 2.0, -0.7]]).astype(np.float32)
>>> F.triplet(anchor, pos, neg)
variable(0.14000003)
>>> y = F.triplet(anchor, pos, neg, reduce='no')
>>> y.shape
(2,)
>>> y.array
array([0.11000005, 0.17      ], dtype=float32)
>>> F.triplet(anchor, pos, neg, margin=0.5)  # harder penalty
variable(0.44000003)
```

4.2.7 Mathematical functions

`chainer.functions.absolute`	Element-wise absolute.
`chainer.functions.arccos`	Elementwise arccosine function.
`chainer.functions.arcsin`	Elementwise arcsine function.
`chainer.functions.arctan`	Elementwise arctangent function.
`chainer.functions.arctan2`	Elementwise arctangent function with two arguments.
`chainer.functions.arctanh`	Elementwise inverse hyperbolic tangent function.
`chainer.functions.argmax`	Returns index which holds maximum of array elements over a given axis.
`chainer.functions.argmin`	Returns index which holds minimum of array elements over a given axis.
`chainer.functions.average`	Calculate weighted average of array elements over a given axis.
`chainer.functions.batch_inv`	Computes the inverse of a batch of square matrices.
`chainer.functions.batch_l2_norm_squared`	L2 norm (a.k.a. Euclidean norm) squared.
`chainer.functions.batch_matmul`	Computes the batch matrix multiplications of two sets of arrays.

Continued on next page

Table 8 – continued from previous page

chainer.functions.bias	Elementwise summation with broadcasting.
chainer.functions.ceil	Elementwise ceil function.
chainer.functions.clip	Clips (limits) elements of input variable.
chainer.functions.cos	Elementwise cos function.
chainer.functions.cosh	Elementwise hyperbolic cosine function.
chainer.functions.cumprod	Cumulative prod of array elements over a given axis.
chainer.functions.cumsum	Cumulative sum of array elements over a given axis.
chainer.functions.det	Computes the determinant of a single square matrix.
chainer.functions.batch_det	Computes the determinant of a batch of square matrices.
chainer.functions.digamma	Digamma function.
chainer.functions.einsum	Einstein summation
chainer.functions.erf	Elementwise error function.
chainer.functions.erfc	Elementwise complementary error function.
chainer.functions.erfcinv	Elementwise inverse function of complementary error function.
chainer.functions.erfcx	Elementwise scaled complementary error function.
chainer.functions.erfinv	Elementwise inverse function of error function.
chainer.functions.exp	Elementwise exponential function.
chainer.functions.expm1	Elementwise exponential minus one function.
chainer.functions.fft	Fast Fourier transform.
chainer.functions.fix	Elementwise fix function.
chainer.functions.fmod	Elementwise mod function.
chainer.functions.floor	Elementwise floor function.
chainer.functions.identity	Just returns input variables.
chainer.functions.ifft	Inverse fast Fourier transform.
chainer.functions.inv	Computes the inverse of square matrix.
chainer.functions.lgamma	logarithm of gamma function.
chainer.functions.linear_interpolate	Elementwise linear-interpolation function.
chainer.functions.log	Elementwise natural logarithm function.
chainer.functions.log10	Elementwise logarithm function to the base 10.
chainer.functions.log1p	Elementwise natural logarithm plus one function.
chainer.functions.log2	Elementwise logarithm function to the base 2.
chainer.functions.log_ndtr	Logarithm of cumulative distribution function of normal distribution.
chainer.functions.logsumexp	Log-sum-exp of array elements over a given axis.
chainer.functions.matmul	Computes the matrix multiplication of two arrays.
chainer.functions.max	Maximum of array elements over a given axis.
chainer.functions.maximum	Element-wise maximum of input variables.
chainer.functions.mean	Calculate weighted average of array elements over a given axis.
chainer.functions.min	Minimum of array elements over a given axis.
chainer.functions.minimum	Element-wise minimum of input variables.
chainer.functions.ndtr	Elementwise cumulative distribution function of normal distribution.
chainer.functions.ndtri	Elementwise inverse function of ndtr.
chainer.functions.prod	Product of array elements over a given axis.
chainer.functions.polygamma	Polygamma function.
chainer.functions.rsqrt	Computes elementwise reciprocal of square root of input x_i.
chainer.functions.scale	Elementwise product with broadcasting.

Continued on next page

Table 8 – continued from previous page

`chainer.functions.sin`	Elementwise sin function.
`chainer.functions.sinh`	Elementwise hyperbolic sine function.
`chainer.functions.sign`	Elementwise sign function.
`chainer.functions.sparse_matmul`	Computes the batched multiplication of sparse and dense matrix.
`chainer.functions.sqrt`	Elementwise square root function.
`chainer.functions.square`	Elementwise square function.
`chainer.functions.squared_difference`	Squared difference function.
`chainer.functions.sum`	Sum of array elements over a given axis.
`chainer.functions.sum_to`	Sum elements along axes to output an array of a given shape.
`chainer.functions.tanh`	Elementwise hyperbolic tangent function.
`chainer.functions.tan`	Elementwise tan function.
`chainer.functions.tensordot`	Returns the tensor dot product of two arrays along specified axes.
`chainer.functions.zeta`	Zeta function.

chainer.functions.absolute

`chainer.functions.absolute(self)`

Element-wise absolute.

Returns Output variable.

Return type *Variable*

chainer.functions.arccos

`chainer.functions.arccos(x)`

Elementwise arccosine function.

$$y_i = \arccos x_i.$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.arcsin

`chainer.functions.arcsin(x)`

Elementwise arcsine function.

$$y_i = \arcsin x_i.$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.arctan

`chainer.functions.arctan(x)`

Elementwise arctangent function.

$$y_i = \arctan x_i.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.arctan2

`chainer.functions.arctan2(x1, x2)`

Elementwise arctangent function with two arguments.

Parameters

- **x1** (*Variable* or *N-dimensional array*) – Y-coordinates.
- **x2** (*Variable* or *N-dimensional array*) – X-coordinates.

Returns Angles in radians, in the range [-pi, pi].

Return type *Variable*

chainer.functions.arctanh

`chainer.functions.arctanh(x)`

Elementwise inverse hyperbolic tangent function.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.argmax

`chainer.functions.argmax(x, axis=None)`

Returns index which holds maximum of array elements over a given axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Array to find maximum elements.
- **axis** (*None or int*) – Axis over which a max is performed. The default (axis = None) is perform a max over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

chainer.functions.argmin

`chainer.functions.argmin(x, axis=None)`

Returns index which holds minimum of array elements over a given axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Array to find minimum elements.
- **axis** (*None or int*) – Axis over which a min is performed. The default (axis = None) is perform a min over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

chainer.functions.average

`chainer.functions.average(x, axis=None, weights=None, keepdims=False)`

Calculate weighted average of array elements over a given axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Elements to sum.
- **axis** (*None or int or tuple of int*) – Axis which the method is performed. With the default (axis = None) it performs a mean over all the dimensions of the input array.
- **weights** (None or *Variable* or *N-dimensional array*) – An array holding weights to calculate weighted average. If it is `None`, all weights are assumed to be one. When `axis` is `None`, `weights` must have the same shape of `x`. And when `axis` is `int`, it must be 1-D array satisfying `weights.shape == (x.shape[axis],)`.
- **keepdims** (*bool*) – If `True`, the specified axes are remained as axes of length one.

Returns Output variable.

Return type *Variable*

chainer.functions.batch_inv

`chainer.functions.batch_inv(a)`

Computes the inverse of a batch of square matrices.

Parameters **a** (*Variable* or *N-dimensional array*) – Input array to compute the inverse for. Shape of the array should be `(m, n, n)` where `m` is the number of matrices in the batch, and `n` is the dimensionality of a square matrix.

Returns Inverse of every matrix in the batch of matrices.

Return type *Variable*

chainer.functions.batch_l2_norm_squared

`chainer.functions.batch_l2_norm_squared(x)`

L2 norm (a.k.a. Euclidean norm) squared.

This function implements the square of L2 norm on a vector. No reduction along batch axis is done.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. The first dimension is assumed to be the *minibatch dimension*. If x has more than two dimensions all but the first dimension are flattened to one dimension.

Returns Two dimensional output variable.

Return type *Variable*

chainer.functions.batch_matmul

chainer.functions.**batch_matmul**(*a*, *b*, *transa=False*, *transb=False*)

Computes the batch matrix multiplications of two sets of arrays.

Parameters

- **a** (*Variable* or *N-dimensional array*) – The left operand of the batch matrix multiplications. A 2-D array of shape (B, N) is considered as B $N \times 1$ matrices. A 3-D array of shape (B, M, N) is considered as B $M \times N$ matrices.
- **b** (*Variable* or *N-dimensional array*) – The right operand of the batch matrix multiplications. Its array is treated as matrices in the same way as a's array.
- **transa** (*bool*) – If True, transpose each matrix in a.
- **transb** (*bool*) – If True, transpose each matrix in b.

Returns The result of the batch matrix multiplications as a 3-D array.

Return type *Variable*

Deprecated since version v3.0.0: batch_matmul is deprecated. Use matmul instead.

chainer.functions.bias

chainer.functions.**bias**(*x*, *y*, *axis=1*)

Elementwise summation with broadcasting.

Computes a elementwise summation of two input variables, with the shape of the latter variable broadcasted to match the shape of the former. axis is the first axis of the first variable along which the second variable is applied.

The term "broadcasting" here comes from Caffe's bias layer so the "broadcasting" with the following arguments:

```
   x : 100 x 3 x 40 x 5 x 6
   y : 3 x 40
axis : 1
```

is equivalent to the following numpy broadcasting:

```
x : 100 x  3 x 40 x 5 x 6
y :   (1 x) 3 x 40 x 1 x 1
```

Note that the axis of x to which we apply y is specified by the argument axis, whose meaning is different from numpy's axis.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable to be summed.
- **y** (*Variable* or *N-dimensional array*) – Input variable to sum, broadcasted.

- **axis** (*int*) – The first axis of `x` along which `y` is applied.

Returns Output variable.

Return type *Variable*

chainer.functions.ceil

`chainer.functions.ceil(x)`

Elementwise ceil function.

$$y_i = \lceil x_i \rceil$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.clip

`chainer.functions.clip(x, x_min, x_max)`

Clips (limits) elements of input variable.

Given an interval `[x_min, xmax]`, elements outside the interval are clipped to the interval edges.

Its gradients at `x_min` and `x_max` are regarded as 1.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable to be clipped.
- **x_min** (*float*) – Minimum value.
- **x_max** (*float*) – Maximum value.

Returns Output variable.

Return type *Variable*

chainer.functions.cos

`chainer.functions.cos(x)`

Elementwise cos function.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.cosh

`chainer.functions.cosh(x)`

Elementwise hyperbolic cosine function.

$$y_i = \cosh x_i.$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.cumprod

`chainer.functions.cumprod(x, axis=None)`
Cumulative prod of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Elements to calculate the cumulative prod.
- **axis** (*int or None*) – Axis along which the cumulative prod is taken. If it is not specified, the input is flattened.

Returns Output variable.

Return type *Variable*

chainer.functions.cumsum

`chainer.functions.cumsum(x, axis=None)`
Cumulative sum of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Elements to calculate the cumulative sum.
- **axis** (*int or None*) – Axis along which the cumulative sum is taken. If it is not specified, the input is flattened.

Returns Output variable.

Return type *Variable*

chainer.functions.det

`chainer.functions.det(a)`
Computes the determinant of a single square matrix.

Parameters **a** (`Variable` or *N-dimensional array*) – Input array to compute the determinant for.

Returns Scalar determinant of the matrix a.

Return type *Variable*

chainer.functions.batch_det

`chainer.functions.batch_det(a)`
Computes the determinant of a batch of square matrices.

Parameters **a** (`Variable` or *N-dimensional array*) – Input array to compute the determinant for. The first dimension should iterate over each matrix and be of the batchsize.

Returns vector of determinants for every matrix in the batch.

Return type *Variable*

chainer.functions.digamma

chainer.functions.**digamma**(*x*)

Digamma function.

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.einsum

chainer.functions.**einsum**(**operands*)

Einstein summation

This function supports two formats of inputs:

- `einsum(subscripts, op0, op1, ...)`
- `einsum(op0, sublist0, op1, sublist1, ..., [sublistout])`

See also `numpy.einsum()`

Example

The following example computes a batched application of a bilinear function with weight `w`.

```
>>> x1 = np.arange(12).reshape(3, 4).astype(np.float32)
>>> x2 = np.arange(15).reshape(3, 5).astype(np.float32)
>>> w = np.arange(120).reshape(4, 5, 6).astype(np.float32)
>>> y = F.einsum('ij,ik,jkl->il', x1, x2, w)
>>> y.shape
(3, 6)
```

The batch axes can be denoted by `...`. If the string of output subscripts is omitted, the summation is taken over the subscript alphabets with two (or more) occurrences.

```
>>> np.allclose(y.array, F.einsum('...j,...k,jkl', x1, x2, w).array)
True
```

In the other format:

```
>>> y = F.einsum(x1, [0, 1], x2, [0, 2], w, [1, 2, 3], [0, 3])
>>> y.shape
(3, 6)
>>> y = F.einsum(x1, [Ellipsis, 1], x2, [Ellipsis, 2], w, [1, 2, 3])
>>> y.shape
(3, 6)
```

chainer.functions.erf

`chainer.functions.erf(x)`

Elementwise error function.

Note: Forward computation in CPU can be slow if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.erfc

`chainer.functions.erfc(x)`

Elementwise complementary error function.

Note: Forward computation in CPU can be slow if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.erfcinv

`chainer.functions.erfcinv(x)`

Elementwise inverse function of complementary error function.

Note: Forward computation in CPU cannot be done if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.erfcx

`chainer.functions.erfcx(x)`

Elementwise scaled complementary error function.

Note: Forward computation in CPU cannot be done if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.erfinv

`chainer.functions.erfinv(x)`
Elementwise inverse function of error function.

Note: Forward computation in CPU cannot be done if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.exp

`chainer.functions.exp(x)`
Elementwise exponential function.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.expm1

`chainer.functions.expm1(x)`
Elementwise exponential minus one function.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.fft

`chainer.functions.fft(x)`
Fast Fourier transform.

Parameters **x** (`tuple`) – `(real, imag)` where `real` is a `Variable` or an *N-dimensional array* storing the real part and `imag` is a `Variable` or an *N-dimensional array* storing the imaginary part.

Returns Returns `(ry, iy)` where `ry` is the real part of the result and `iy` is the imaginary part of the result.

Return type tuple

Note: Currently this function supports a tuple as input. It will support a complex numbers directly in the future.

chainer.functions.fix

`chainer.functions.fix(x)`

Elementwise fix function.

$$y_i = x_i$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.fmod

`chainer.functions.fmod(x, divisor)`

Elementwise mod function.

$$y_i = x_i \bmod \mathrm{divisor}.$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **divisor** (*Variable* or *N-dimensional array*) – Input divisor.

Returns Output variable.

Return type *Variable*

chainer.functions.floor

`chainer.functions.floor(x)`

Elementwise floor function.

$$y_i = \lfloor x_i \rfloor$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.identity

`chainer.functions.identity(*inputs)`

Just returns input variables.

chainer.functions.ifft

`chainer.functions.ifft(x)`

Inverse fast Fourier transform.

Parameters **x** (*tuple*) – `(real, imag)` where `real` is a *Variable* or an *N-dimensional array* storing the real part and `imag` is a *Variable* or an *N-dimensional array* storing the imaginary part.

Returns Returns `(ry, iy)` where `ry` is the real part of the result and `iy` is the imaginary part of the result.

Return type tuple

Note: Currently this function supports a tuple as input. It will support a complex numbers directly in the future.

chainer.functions.inv

`chainer.functions.inv(a)`

Computes the inverse of square matrix.

a (*Variable* or *N-dimensional array*): Input array to compute the inverse for. Shape of the array should be `(n, n)` where `n` is the dimensionality of a square matrix.

Returns Matrix inverse of `a`.

Return type *Variable*

chainer.functions.lgamma

`chainer.functions.lgamma(x)`

logarithm of gamma function.

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.linear_interpolate

`chainer.functions.linear_interpolate(p, x, y)`

Elementwise linear-interpolation function.

This function is defined as

$$f(p, x, y) = px + (1 - p)y.$$

Parameters

- **p** (*Variable* or *N-dimensional array*) – Input variable.
- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **y** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.log

`chainer.functions.log(x)`

Elementwise natural logarithm function.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.log10

`chainer.functions.log10(x)`

Elementwise logarithm function to the base 10.

$$y_i = \log_{10} x_i.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.log1p

`chainer.functions.log1p(x)`

Elementwise natural logarithm plus one function.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.log2

`chainer.functions.log2(x)`

Elementwise logarithm function to the base 2.

$$y_i = \log_2 x_i.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.log_ndtr

`chainer.functions.log_ndtr(x)`

Logarithm of cumulative distribution function of normal distribution.

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.logsumexp

`chainer.functions.logsumexp(x, axis=None)`

Log-sum-exp of array elements over a given axis.

This function calculates logarithm of sum of exponential of array elements.

$$y_i = \log \left(\sum_j \exp(x_{ij}) \right)$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Elements to log-sum-exp.
- **axis** (*None, int, or tuple of int*) – Axis which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

chainer.functions.matmul

`chainer.functions.matmul(a, b, transa=False, transb=False)`

Computes the matrix multiplication of two arrays.

Parameters

- **a** (*Variable* or *N-dimensional array*) – The left operand of the matrix multiplication. If `a` and `b` are both 1-D arrays, `matmul` returns a dot product of vector *a* and vector *b*. If 2-D arrays, `matmul` returns matrix product of `a` and `b`. If either's dimension is larger than 2, they are treated as a stack of matrices residing in the last two indexes. `matmul` returns a stack of each two arrays. In this case, `a` and `b` are broadcasted along axes except the last two.
- **b** (*Variable* or *N-dimensional array*) – The right operand of the matrix multiplication. Its array is treated as a matrix in the same way as `a`'s array.
- **transa** (*bool*) – If `True`, each matrices in `a` will be transposed. If `a.ndim == 1`, do nothing.
- **transb** (*bool*) – If `True`, each matrices in `b` will be transposed. If `b.ndim == 1`, do nothing.

Returns The result of the matrix multiplication.

Return type *Variable*

Example

```
>>> a = np.array([[1, 0], [0, 1]], np.float32)
>>> b = np.array([[4, 1], [2, 2]], np.float32)
>>> F.matmul(a, b).array
array([[4., 1.],
       [2., 2.]], dtype=float32)
```

chainer.functions.max

`chainer.functions.max(x, axis=None, keepdims=False)`

Maximum of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Array to be maximized.
- **axis** (`None, int, or tuple of int`) – Axis over which a max is performed. The default (axis = None) is perform a max over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

chainer.functions.maximum

`chainer.functions.maximum(x1, x2)`

Element-wise maximum of input variables.

Parameters

- **x1** (`Variable` or *N-dimensional array*) – Input variables to be compared. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **x2** (`Variable` or *N-dimensional array*) – Input variables to be compared. A $(s_1, s_2, ..., s_N)$ -shaped float array.

Returns Output variable.

Return type *Variable*

Example

```
>>> x1 = np.arange(6).astype(np.float32)
>>> x1
array([0., 1., 2., 3., 4., 5.], dtype=float32)
>>> x2 = np.array([5, 4, 3, 2, 1, 0]).astype(np.float32)
>>> x2
array([5., 4., 3., 2., 1., 0.], dtype=float32)
>>> y = F.maximum(x1, x2)
>>> y.shape
(6,)
>>> y.array
array([5., 4., 3., 3., 4., 5.], dtype=float32)
```

chainer.functions.mean

`chainer.functions.mean(x, axis=None, weights=None, keepdims=False)`

Calculate weighted average of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Elements to sum.
- **axis** (`None or int or tuple of int`) – Axis which the method is performed. With the default (axis = None) it performs a mean over all the dimensions of the input array.
- **weights** (None or `Variable` or *N-dimensional array*) – An array holding weights to calculate weighted average. If it is `None`, all weights are assumed to be one. When `axis` is `None`, `weights` must have the same shape of `x`. And when `axis` is `int`, it must be 1-D array satisfying `weights.shape == (x.shape[axis],)`.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.

Returns Output variable.

Return type *Variable*

chainer.functions.min

`chainer.functions.min(x, axis=None, keepdims=False)`

Minimum of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Array to be minimized.
- **axis** (`None, int, or tuple of int`) – Axis over which a min is performed. The default (axis = None) is perform a min over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

chainer.functions.minimum

`chainer.functions.minimum(x1, x2)`

Element-wise minimum of input variables.

Parameters

- **x1** (`Variable` or *N-dimensional array*) – Input variables to be compared.
- **x2** (`Variable` or *N-dimensional array*) – Input variables to be compared.

Returns Output variable.

Return type *Variable*

chainer.functions.ndtr

`chainer.functions.ndtr(x)`

Elementwise cumulative distribution function of normal distribution.

Note: Forward computation in CPU can be slow if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.ndtri

`chainer.functions.ndtri(x)`

Elementwise inverse function of ndtr.

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.prod

`chainer.functions.prod(x, axis=None, keepdims=False)`

Product of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Elements to calculate the product.
- **axis** (`None, int, or tuple of int`) – Axis which a product is performed. The default (axis = None) is perform a product over all the dimensions of the input array.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.

Returns Output variable.

Return type *Variable*

chainer.functions.polygamma

`chainer.functions.polygamma(n, x)`

Polygamma function.

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters

- **n** (`Variable` or *N-dimensional array*) – Input variable.
- **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.rsqrt

`chainer.functions.rsqrt(x)`

Computes elementwise reciprocal of square root of input x_i.

$$y_i = \frac{1}{\sqrt{x_i}}.$$

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

See also:

`sqrt()`

chainer.functions.scale

`chainer.functions.scale(x, y, axis=1)`

Elementwise product with broadcasting.

Computes a elementwise product of two input variables, with the shape of the latter variable broadcasted to match the shape of the former. `axis` is the first axis of the first variable along which the second variable is applied.

The term "broadcasting" here comes from Caffe's scale layer so the "broadcasting" with the following arguments:

```
   x : 100 x 3 x 40 x 5 x 6
   y : 3 x 40
axis : 1
```

is equivalent to the following numpy broadcasting:

```
x : 100 x  3 x 40 x 5 x 6
y :  (1 x) 3 x 40 x 1 x 1
```

Note that the axis of `x` to which we apply `y` is specified by the argument `axis`, whose meaning is different from numpy's `axis`.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable to be scaled.
- **y** (`Variable` or *N-dimensional array*) – Input variable to scale, broadcasted.
- **axis** (`int`) – The first axis of `x` along which `y` is applied.

Returns Output variable.

Return type *Variable*

chainer.functions.sin

`chainer.functions.sin(x)`

Elementwise sin function.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.sinh

`chainer.functions.sinh(x)`

Elementwise hyperbolic sine function.

$$y_i = \sinh x_i.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.sign

`chainer.functions.sign(x)`

Elementwise sign function.

For a given input x, this function returns $sgn(x)$ defined as

$$sgn(x) = \begin{cases} -1 & \text{if x} < 0 \\ 0 & \text{if x} = 0 \\ 1 & \text{if x} > 0 \end{cases}$$

Note: The gradient of this function is `None` everywhere and therefore unchains the computational graph.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable for which the sign is computed.

Returns Output variable.

Return type *Variable*

chainer.functions.sparse_matmul

`chainer.functions.sparse_matmul(a, b, transa=False, transb=False)`

Computes the batched multiplication of sparse and dense matrix.

The following use cases are supported:

1. C (dense) = A (sparse) * B (dense)
2. C (dense) = A (dense) * B (sparse)

Parameters

- **a** (*Variable or CooMatrix*) – The left operand of matrix multiplication.

- **b** (Variable *or* CooMatrix) – The right operand of matrix multiplication.
- **transa** (*bool*) – If True, each matrix in a will be transposed.
- **transb** (*bool*) – If True, each matrix in b will be transposed.

Returns Result of batched mat-mul.

Return type *Variable*

See also:

See *to_coo()* for how to construct a COO matrix from an array.

Note: Performance of this function on GPU can be improved by using the order argument of *CooMatrix* when the sparse matrix is created.

chainer.functions.sqrt

chainer.functions.**sqrt**(*x*)

Elementwise square root function.

$$y_i = \sqrt{x}_i.$$

If the value of x_i is negative, it returns Nan for y_i respect to underlying numpy and cupy specification.

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.square

chainer.functions.**square**(*x*)

Elementwise square function.

$$y_i = x_i^2.$$

Parameters **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$ -shaped float array.

Returns Output variable. A $(s_1, s_2, ..., s_N)$ -shaped float array.

Return type *Variable*

Example

```
>>> x = np.arange(6).reshape(2,3).astype(np.float32)
>>> x
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
>>> y = F.square(x)
>>> y.shape
(2, 3)
>>> y.array
array([[ 0.,  1.,  4.],
       [ 9., 16., 25.]], dtype=float32)
```

chainer.functions.squared_difference

`chainer.functions.`**`squared_difference`**(*x1*, *x2*)

Squared difference function.

This functions is identical to `squared_error()` except for the names of the arguments.

See also:

`squared_error()`

chainer.functions.sum

`chainer.functions.`**`sum`**(*x*, *axis=None*, *keepdims=False*)

Sum of array elements over a given axis.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Elements to sum. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **axis** (`None, int, or tuple of int`) – Axis along which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.

Returns Output variable.

Return type *Variable*

Example

```
>>> x = np.arange(6).reshape(2,3).astype(np.float32)
>>> x
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
>>> y = F.sum(x)
>>> y.shape
()
>>> y.array
array(15., dtype=float32)
>>> y = F.sum(x, axis=1)
>>> y.shape
(2,)
>>> y.array
array([ 3., 12.], dtype=float32)
>>> y = F.sum(x, keepdims=True)
>>> y.shape
(1, 1)
>>> y.array
array([[15.]], dtype=float32)
```

chainer.functions.sum_to

`chainer.functions.sum_to(x, shape)`

Sum elements along axes to output an array of a given shape.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **shape** (`tuple of int`) – The target shape.

Returns Output variable of shape `shape`.

Return type *Variable*

Example

```
>>> x = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> x
array([[1., 2., 3.],
       [4., 5., 6.]])
>>> y = F.sum_to(x, (1, 3))
>>> y
variable([[5., 7., 9.]])
>>> z = F.sum_to(x, (2, 1))
>>> z
variable([[ 6.],
          [15.]])
```

---

chainer.functions.tan

`chainer.functions.tan(x)`

Elementwise tan function.

Parameters **x** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

chainer.functions.tensordot

`chainer.functions.tensordot(a, b, axes=2)`

Returns the tensor dot product of two arrays along specified axes.

This is equivalent to compute dot product along the specified axes which are treated as one axis by reshaping.

Parameters

- **a** (`Variable` or *N-dimensional array*) – The first argument.
- **b** (`Variable` or *N-dimensional array*) – The second argument.
- **axes** –
  - If it is an integer, then `axes` axes at the last of `a` and the first of `b` are used.
  - If it is a pair of sequences of integers, then these two sequences specify the list of axes for `a` and `b`. The corresponding axes are paired for sum-product.

Returns The tensor dot product of `a` and `b` along the axes specified by `axes`.

Return type *Variable*

Example

```
>>> a = np.random.rand(5, 3, 2)
>>> b = np.random.rand(3, 2, 4)
>>> c = F.tensordot(a, b, axes=2)
>>> c.shape
(5, 4)
```

See also:

`numpy.tensordot()`

chainer.functions.zeta

`chainer.functions.zeta(x, q)`

Zeta function.

Differentiable only with respect to q

Note: Forward computation in CPU can not be done if SciPy is not available.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Input variable.
- **q** (`Variable` or *N-dimensional array*) – Input variable.

Returns Output variable.

Return type *Variable*

4.2.8 Noise injections

chainer.functions.dropout	Drops elements of input variable randomly.
chainer.functions.gaussian	Gaussian sampling function.
chainer.functions.gumbel_softmax	Gumbel-Softmax sampling function.
chainer.functions.simplified_dropconnect	Linear unit regularized by simplified dropconnect.
chainer.functions.zoneout	Drops elements of input variable and sets to previous variable randomly.

chainer.functions.dropout

`chainer.functions.dropout(x, ratio=.5, *, mask=None, return_mask=False)`

Drops elements of input variable randomly.

This function drops input elements randomly with probability `ratio` and scales the remaining elements by factor `1 / (1 - ratio)`. In testing mode (i.e., `chainer.config.train` is set to `False`), it does nothing and just returns `x`.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. A $(s_1, s_2, ..., s_N)$ -shaped float array.
- **ratio** (*float*) – Dropout ratio. The `ratio` must be `0.0 <= ratio < 1.0`.
- **mask** (*N-dimensional array* or None) – The mask to be used for dropout. You do not have to specify this value, unless you need to make results deterministic. If `mask` is not specified or set to `None`, a mask will be generated randomly according to the given `ratio`. If `mask` is specified, `ratio` will be ignored. The shape and dtype must be the same as `x` and should be on the same device. Note that iDeep and cuDNN will not be used for this function if mask is specified, as iDeep and cuDNN do not support it.
- **return_mask** (*bool*) – If `True`, the mask used for dropout is returned together with the output variable. The returned mask can later be reused by passing it to `mask` argument.

Returns When `return_mask` is `False` (default), returns the output variable. When `True`, returns the tuple of the output variable and mask (*N-dimensional array*). The mask will be on the same device as the input. The mask will become `None` when `chainer.config.train` is set to `False`.

Return type *Variable* or tuple

See the paper by G. Hinton: Improving neural networks by preventing co-adaptation of feature detectors.

Example

```
>>> x = np.array([[-1, 0], [2, -3], [-2, 1]], np.float32)
>>> with chainer.using_config('train', True):
...     y = F.dropout(x)
>>> y.array
array([[-2.,  0.],
       [ 4., -6.],
       [-0.,  2.]], dtype=float32)
>>> with chainer.using_config('train', True):
...     y = F.dropout(x, ratio=0.0) # dropout returns original input if ratio=0.0
>>> (x == y.array).all()
True
>>> with chainer.using_config('train', False):
...     y = F.dropout(x) # dropout in test mode returns original input
>>> (x == y.array).all()
True
```

chainer.functions.gaussian

`chainer.functions.gaussian(mean, ln_var, *, eps=None, return_eps=False)`

Gaussian sampling function.

This function takes a mean μ and the logarithm of a variance $\log(\sigma^2)$ as inputs and outputs a sample drawn from a Gaussian distribution $N(\mu, \sigma)$.

The inputs must have the same shape.

Parameters

- **mean** (*Variable* or *N-dimensional array*) – Input variable representing the mean μ.

- **ln_var** (*Variable* or *N-dimensional array*) – Input variable representing the logarithm of a variance $\log(\sigma^2)$.
- **eps** (*N-dimensional array* or None) – The eps value to be used. You do not have to specify this value, unless you need to make results deterministic. If `eps` is not specified or set to `None`, an eps value will be generated randomly. The shape and dtype must be the same as `ln_var` and should be on the same device.
- **return_eps** (*bool*) – If `True`, the eps value used in this function is returned together with the output variable. The returned eps can later be reused by passing it to the `eps` argument.

Returns When `return_eps` is `False` (default), returns the output variable with the shape of `mean` and/or `ln_var`. When `True`, returns the tuple of the output variable and eps (*N-dimensional array*). The eps will be on the same device as the input (`ln_var`).

Return type *Variable* or tuple

chainer.functions.gumbel_softmax

`chainer.functions.`**`gumbel_softmax`**(*log_pi*, *tau=0.1*, *axis=1*)

Gumbel-Softmax sampling function.

This function draws samples y_i from Gumbel-Softmax distribution,

$$y_i = \frac{\exp((g_i + \log\pi_i)/\tau)}{\sum_j \exp((g_j + \log\pi_j)/\tau)},$$

where τ is a temperature parameter and g_i s are samples drawn from Gumbel distribution $Gumbel(0, 1)$

See Categorical Reparameterization with Gumbel-Softmax.

Parameters

- **log_pi** (*Variable* or *N-dimensional array*) – Input variable representing pre-normalized log-probability $\log\pi$.
- **tau** (`float` or *Variable* or *N-dimensional array*) – Input variable representing temperature τ.

Returns Output variable.

Return type *Variable*

chainer.functions.simplified_dropconnect

`chainer.functions.`**`simplified_dropconnect`**(*x*, *W*, *b=None*, *ratio=0.5*, *train=True*, *mask=None*, *use_batchwise_mask=True*)

Linear unit regularized by simplified dropconnect.

Simplified dropconnect drops weight matrix elements randomly with probability `ratio` and scales the remaining elements by factor `1 / (1 - ratio)`. It accepts two or three arguments: an input minibatch `x`, a weight matrix `W`, and optionally a bias vector `b`. It computes $Y = xW^\top + b$.

In testing mode, zero will be used as simplified dropconnect ratio instead of `ratio`.

Notice: This implementation cannot be used for reproduction of the paper. There is a difference between the current implementation and the original one. The original version uses sampling with gaussian distribution before passing activation function, whereas the current implementation averages before activation.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable. Its first dimension `n` is assumed to be the *minibatch dimension*. The other dimensions are treated as concatenated one dimension whose size must be `N`.
- **W** (*Variable* or *N-dimensional array*) – Weight variable of shape `(M, N)`.
- **b** (*Variable* or *N-dimensional array*) – Bias variable (optional) of shape `(M,)`.
- **ratio** (*float*) – Dropconnect ratio.
- **train** (*bool*) – If `True`, executes simplified dropconnect. Otherwise, simplified dropconnect function works as a linear function.
- **mask** (None or *Variable* or *N-dimensional array*) – If `None`, randomized dropconnect mask is generated. Otherwise, The mask must be `(n, M, N)` or `(M, N)` shaped array, and *use_batchwise_mask* is ignored. Main purpose of this option is debugging. *mask* array will be used as a dropconnect mask.
- **use_batchwise_mask** (*bool*) – If `True`, dropped connections depend on each sample in mini-batch.

Returns Output variable.

Return type *Variable*

See also:

`Dropconnect`

See also:

Li, W., Matthew Z., Sixin Z., Yann L., Rob F. (2013). Regularization of Neural Network using DropConnect. International Conference on Machine Learning. URL

chainer.functions.zoneout

`chainer.functions.zoneout`(*h*, *x*, *ratio=.5*)

Drops elements of input variable and sets to previous variable randomly.

This function drops input elements randomly with probability `ratio` and instead sets dropping element to their previous variable. In testing mode , it does nothing and just returns `x`.

Parameters

- **h** (*Variable* or *N-dimensional array*) – Previous variable.
- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **ratio** (*float*) – Zoneout ratio.

Returns Output variable.

Return type *Variable*

See the paper: Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations.

4.2.9 Normalization functions

chainer.functions.batch_normalization	Batch normalization function.
chainer.functions.batch_renormalization	Batch renormalization function.

Continued on next page

Table 10 – continued from previous page

chainer.functions.decorrelated_batch_normalization	Decorrelated batch normalization function.
chainer.functions.fixed_batch_normalization	Batch normalization function with fixed statistics.
chainer.functions.fixed_batch_renormalization	
chainer.functions.fixed_decorrelated_batch_normalization	Decorrelated batch normalization function with fixed statistics.
chainer.functions.group_normalization	Group normalization function.
chainer.functions.layer_normalization	Layer normalization.
chainer.functions.local_response_normalization	Local response normalization across neighboring channels.
chainer.functions.normalize	Normalize input by L2 norm.

chainer.functions.batch_normalization

chainer.functions.**batch_normalization**(*x*, *gamma*, *beta*, *eps=2e-5*, *running_mean=None*, *running_var=None*, *decay=0.9*, *axis=None*)

Batch normalization function.

It takes the input variable x and two parameter variables gamma and beta. The parameter variables must both have the same dimensionality, which is referred to as the channel shape. This channel shape corresponds to the dimensions in the input which are not averaged over. Since the first dimension of the input corresponds to the batch size, the second dimension of x will correspond to the first dimension of the channel shape, the third dimension of x will correspond to the second channel dimension (if it exists) and so on. Therefore, the dimensionality of the input must be at least one plus the number of channel dimensions. The total effective "batch size" will then be considered to be the product of all dimensions in x except for the channel dimensions.

As an example, if the input is four dimensional and the parameter variables are one dimensional, then it is assumed that the first dimension of the input is the batch size, the second dimension is the channel size, and the remaining two dimensions are considered to be spatial dimensions that will be averaged over along with the batch size in the batch normalization computations. That is, the total batch size will be considered to be the product of all input dimensions except the second dimension.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **gamma** (*Variable* or *N-dimensional array*) – Scaling parameter of normalized data.
- **beta** (*Variable* or *N-dimensional array*) – Shifting parameter of scaled normalized data.
- **eps** (*float*) – Epsilon value for numerical stability.
- **running_mean** (*N-dimensional array*) – Running average of the mean. This is a running average of the mean over several mini-batches using the decay parameter. The function takes a previous running average, and updates the array in-place by the new running average. If None, the running average is not computed. If this is None, then runnng_var must also be None.
- **running_var** (*N-dimensional array*) – Running average of the variance. This is a running average of the variance over several mini-batches using the decay parameter. The function takes a previous running average, and updates the array in-place by the new running average. If None, the running average is not computed. If this is None, then running_mean must also be None.
- **decay** (*float*) – Decay rate of moving average. It is used during training.

- **axis** (*int, tuple of int or None*) – Axis over which normalization is performed. When axis is `None`, it is determined from input dimensions. For example, if `x.ndim` is 4, axis becomes (0, 2, 3) and normalization is performed over 0th, 2nd and 3rd axis of input. If it is 2, axis becomes (0) and normalization is performed over 0th axis of input. When a tuple of int is given to this option, numbers in the tuple must be being sorted in ascending order. For example, (0, 2) is OK, but (2, 0) is not.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

See also:

BatchNormalization to manage the model parameters (`gamma`, `beta`) and the statistics (`running_mean`, `running_var`).

chainer.functions.batch_renormalization

`chainer.functions.`**`batch_renormalization`**(*x, gamma, beta, rmax, dmax, eps=2e-05, running_mean=None, running_var=None, decay=0.9, update_statistics=False*)

Batch renormalization function.

This is an extension of batch normalization, which ensures that the training and inference models generate the same outputs that depend on individual examples rather than the entire minibatch.

Note: This function does not perform in-place update to `running_mean` and `running_var` by default, contrary to *batch_normalization()*. If the function is called, it will not be possible to access the updated running mean and variance statistics, because they are members of the function object, which cannot be accessed by the caller. If it is desired to update the running statistics, call the function with `update_statistics=True` option.

Note: For the consistency with Batch Normalization, this function intentionally ignores some of the theoretical flaws in Algorithm 1 of the Batch Renormalization paper:

- `F.batch_renormalization` maintains the moving average of variances σ^2, while the original paper maintains the moving average of standard deviations σ.
- `F.batch_renormalization` applies Bessel's correction to update the moving average of variances.

See: Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models

See also:

BatchRenormalization to manage the model parameters (`gamma`, `beta`) and the statistics (`running_mean`, `running_var`).

chainer.functions.decorrelated_batch_normalization

`chainer.functions.`**`decorrelated_batch_normalization`**(*x, *, groups=16, eps=2e-5, running_mean=None, running_projection=None, decay=0.9*)

Decorrelated batch normalization function.

It takes the input variable `x` and normalizes it using batch statistics to make the output zero-mean and decorrelated.

Parameters

- **x** (*Variable*) – Input variable.
- **groups** (*int*) – Number of groups to use for group whitening.
- **eps** (*float*) – Epsilon value for numerical stability.
- **running_mean** (*N-dimensional array*) – Expected value of the mean. This is a running average of the mean over several mini-batches using the decay parameter. If `None`, the expected mean is initialized to zero.
- **running_projection** (*N-dimensional array*) – Expected value of the project matrix. This is a running average of the projection over several mini-batches using the decay parameter. If `None`, the expected projected is initialized to the identity matrix.
- **decay** (*float*) – Decay rate of moving average. It is used during training.

Returns The output variable which has the same shape as x.

Return type *Variable*

See: Decorrelated Batch Normalization

See also:

DecorrelatedBatchNormalization

chainer.functions.fixed_batch_normalization

`chainer.functions.`**`fixed_batch_normalization`**(*x*, *gamma*, *beta*, *mean*, *var*, *eps=2e-05*, *axis=None*)

Batch normalization function with fixed statistics.

This is a variant of batch normalization, where the mean and variance statistics are given by the caller as fixed variables. This is used on testing mode of the batch normalization layer, where batch statistics cannot be used for prediction consistency.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **gamma** (*Variable* or *N-dimensional array*) – Scaling parameter of normalized data.
- **beta** (*Variable* or *N-dimensional array*) – Shifting parameter of scaled normalized data.
- **mean** (*Variable* or *N-dimensional array*) – Shifting parameter of input.
- **var** (*Variable* or *N-dimensional array*) – Square of scaling parameter of input.
- **eps** (*float*) – Epsilon value for numerical stability.
- **axis** (*int, tuple of int or None*) – Axis over which normalization is performed. When axis is `None`, it is determined from input dimensions. For example, if `x.ndim is 4`, axis becomes (0, 2, 3) and normalization is performed over 0th, 2nd and 3rd axis of input. If it is 2, axis becomes (0) and normalization is performed over 0th axis of input. When a tuple of int is given to this option, numbers in the tuple must be being sorted in ascending order. For example, (0, 2) is OK, but (2, 0) is not.

See also:

batch_normalization(), *BatchNormalization*

chainer.functions.fixed_batch_renormalization

`chainer.functions.fixed_batch_renormalization(x, gamma, beta, mean, var, eps=2e-05)`

chainer.functions.fixed_decorrelated_batch_normalization

`chainer.functions.fixed_decorrelated_batch_normalization(x, mean, projection, groups=16)`

Decorrelated batch normalization function with fixed statistics.

This is a variant of decorrelated batch normalization, where the mean and projection statistics are given by the caller as fixed variables. This is used in testing mode of the decorrelated batch normalization layer, where batch statistics cannot be used for prediction consistency.

Parameters

- **x** (`Variable`) – Input variable.
- **mean** (`Variable` or *N-dimensional array*) – Shifting parameter of input.
- **projection** (`Variable` or *N-dimensional array*) – Projection matrix for decorrelation of input.
- **groups** (`int`) – Number of groups to use for group whitening.

Returns The output variable which has the same shape as x.

Return type *Variable*

See also:

`decorrelated_batch_normalization()`, `DecorrelatedBatchNormalization`

chainer.functions.group_normalization

`chainer.functions.group_normalization(x, groups, gamma, beta, eps=1e-05)`

Group normalization function.

This function implements a "group normalization" which divides the channels into groups and computes within each group the mean and variance, then normalize by these statistics, scales and shifts them.

Parameters

- **x** (`Variable` or *N-dimensional array*) – Batch tensors. First dimension of this value must be the size of minibatch and second dimension must be the number of channels. Moreover, this value must have one or more following dimensions, such as height and width.
- **groups** (`int`) – The number of channel groups. This value must be a divisor of the number of channels.
- **gamma** (`Variable` or *N-dimensional array*) – Scaling parameter.
- **beta** (`Variable` or *N-dimensional array*) – Shifting parameter.
- **eps** (`float`) – Epsilon value for numerical stability of normalization.

Returns The output variable which has the same shape as x.

Return type *Variable*

See: Group Normalization

See also:

GroupNormalization to manage the model parameters gamma and beta.

chainer.functions.layer_normalization

chainer.functions.**layer_normalization**(*x*, *gamma*, *beta*, *eps=1e-05*)

Layer normalization.

This function implements a "layer normalization" which normalizes the input units by statistics that are computed along the second axis, scales and shifts them.

Parameters

- **x** (*Variable* or *N-dimensional array*) – Batch vectors. Shape of this value must be *(batch_size, unit_size)*, e.g., the output of *linear()*.
- **gamma** (*Variable* or *N-dimensional array*) – Scaling vectors.
- **beta** (*Variable* or *N-dimensional array*) – Shifting vectors.

Returns The output variable which has the same shape as x.

Return type *Variable*

See: Layer Normalization

See also:

LayerNormalization to manage the model parameters gamma and beta.

chainer.functions.local_response_normalization

chainer.functions.**local_response_normalization**(*x*, *n=5*, *k=2*, *alpha=0.0001*, *beta=0.75*)

Local response normalization across neighboring channels.

This function implements normalization across channels. Let x an input image with N channels. Then, this function computes an output image y by following formula:

$$y_i = \frac{x_i}{\left(k + \ \alpha \sum_{j=\max 1, i-n/2}^{\min N, i+n/2} \ x_j^2\right)^\beta}.$$

Parameters

- **x** (*Variable* or *N-dimensional array*) – Input variable.
- **n** (*int*) – Normalization window width.
- **k** (*float*) – Smoothing parameter.
- **alpha** (*float*) – Normalizer scaling parameter.
- **beta** (*float*) – Normalizer power parameter.

Returns Output variable.

Return type *Variable*

See: Section 3.3 of ImageNet Classification with Deep Convolutional Neural Networks

chainer.functions.normalize

`chainer.functions.normalize(x, eps=1e-05, axis=1)`

Normalize input by L2 norm.

This function implements L2 normalization on a sample along the given axis/axes. No reduction is done along the normalization axis.

In the case when `axis=1` and $\mathbf{x}$ is a matrix of dimension (N, K), where N and K denote mini-batch size and the dimension of the input vectors, this function computes an output matrix $\mathbf{y}$ of dimension (N, K) by the following equation:

$$\mathbf{y}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|_2 + \epsilon}$$

`eps` is used to avoid division by zero when norm of $\mathbf{x}$ along the given axis is zero.

The default value of `axis` is determined for backward compatibility.

Parameters

- **x** (`Variable` or *N-dimensional array*) – multi-dimensional output variable. The first dimension is assumed to be the mini-batch dimension.
- **eps** (`float`) – Epsilon value for numerical stability.
- **axis** (`int or tuple of ints`) – Axis along which to normalize.

Returns The output variable which has the same shape as x.

Return type *Variable*

4.2.10 Spatial pooling

`chainer.functions.average_pooling_1d`	1-dimensional spatial average pooling function.
`chainer.functions.average_pooling_2d`	Spatial average pooling function.
`chainer.functions.average_pooling_3d`	3-dimensional spatial average pooling function.
`chainer.functions.average_pooling_nd`	N-dimensionally spatial average pooling function.
`chainer.functions.max_pooling_1d`	1-dimensional spatial max pooling function.
`chainer.functions.max_pooling_2d`	Spatial max pooling function.
`chainer.functions.max_pooling_3d`	3-dimensional spatial max pooling function.
`chainer.functions.max_pooling_nd`	N-dimensionally spatial max pooling function.
`chainer.functions.roi_average_align_2d`	Spatial Region of Interest (ROI) average align function.
`chainer.functions.roi_average_pooling_2d`	Spatial Region of Interest (ROI) average pooling function.
`chainer.functions.roi_max_align_2d`	Spatial Region of Interest (ROI) max align function.
`chainer.functions.roi_max_pooling_2d`	Spatial Region of Interest (ROI) max pooling function.
`chainer.functions.roi_pooling_2d`	Spatial Region of Interest (ROI) pooling function.
`chainer.functions.spatial_pyramid_pooling_2d`	Spatial pyramid pooling function.
`chainer.functions.unpooling_1d`	Inverse operation of 1-dimensional spatial pooling.
`chainer.functions.unpooling_2d`	Inverse operation of pooling for 2d array.
`chainer.functions.unpooling_3d`	Inverse operation of 3-dimensional spatial pooling.
`chainer.functions.unpooling_nd`	Inverse operation of N-dimensional spatial pooling.
`chainer.functions.upsampling_2d`	Upsampling using pooling indices.

chainer.functions.average_pooling_1d

`chainer.functions.average_pooling_1d(x, ksize, stride=None, pad=0, pad_value=0)`

1-dimensional spatial average pooling function.

> **Warning:** This feature is experimental. The interface can change in the future.

> **Note:** This function calls *average_pooling_nd()* internally, so see the details of the behavior in the documentation of *average_pooling_nd()*.

chainer.functions.average_pooling_2d

`chainer.functions.average_pooling_2d(x, ksize, stride=None, pad=0)`

Spatial average pooling function.

This function acts similarly to *convolution_2d()*, but it computes the average of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

- **x** (Variable) – Input variable.
- **ksize** (*int or pair of ints*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints or None*) – Stride of pooling applications. `stride=s` and `stride=(s, s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or pair of ints*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p)` are equivalent.

Returns Output variable.

Return type *Variable*

> **Note:** This function currently does not support `cover_all` mode as *max_pooling_2d()*. Average pooling runs in non-cover-all mode.

> **Note:** The values in the padded region is treated as 0, leading the averages biased towards zero. To obtain unbiased averages, use *average_pooling_nd()* with `pad_value=None`.

chainer.functions.average_pooling_3d

`chainer.functions.average_pooling_3d(x, ksize, stride=None, pad=0, pad_value=0)`

3-dimensional spatial average pooling function.

> **Warning:** This feature is experimental. The interface can change in the future.

Note: This function calls `average_pooling_nd()` internally, so see the details of the behavior in the documentation of `average_pooling_nd()`.

chainer.functions.average_pooling_nd

`chainer.functions.average_pooling_nd`(*x*, *ksize*, *stride=None*, *pad=0*, *pad_value=0*)

N-dimensionally spatial average pooling function.

Warning: This feature is experimental. The interface can change in the future.

This function provides a N-dimensionally generalized version of `average_pooling_2d()`. This acts similarly to `convolution_nd()`, but it computes the average of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

- **x** (`Variable`) – Input variable.
- **ksize** (*int or tuple of ints*) – Size of pooling window. `ksize=k` and `ksize=(k, k, ..., k)` are equivalent.
- **stride** (*int or tuple of ints or None*) – Stride of pooling applications. `stride=s` and `stride=(s, s, ..., s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or tuple of ints*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p, ..., p)` are equivalent.
- **pad_value** (*0 or None*) – Value to fill the padded region when calculating average. If `None` is specified, such region is ignored. The default value is `0`, therefore the averages are biased towards zero.

Returns Output variable.

Return type *Variable*

Note: This function currently does not support `cover_all` mode as `max_pooling_nd()`. Average pooling runs in non-cover-all mode.

chainer.functions.max_pooling_1d

`chainer.functions.max_pooling_1d`(*x*, *ksize*, *stride=None*, *pad=0*, *cover_all=True*, *return_indices=False*)

1-dimensional spatial max pooling function.

Warning: This feature is experimental. The interface can change in the future.

Note: This function calls `max_pooling_nd()` internally, so see the details of the behavior in the documentation of `max_pooling_nd()`.

chainer.functions.max_pooling_2d

`chainer.functions.max_pooling_2d(x, ksize, stride=None, pad=0, cover_all=True, return_indices=False)`

Spatial max pooling function.

This function acts similarly to `convolution_2d()`, but it computes the maximum of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

- **x** (`Variable`) – Input variable.
- **ksize** (*int or pair of ints*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints or None*) – Stride of pooling applications. `stride=s` and `stride=(s, s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or pair of ints*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p)` are equivalent.
- **cover_all** (*bool*) – If `True`, all spatial locations are pooled into some output pixels. It may make the output size larger.
- **return_indices** (*bool*) – If `True`, pooling indices array is returned together with the output variable. The returned indices are expected for use by `chainer.functions.upsampling_2d()`. Note that cuDNN will not be used for this function if `return_indices` is set to `True`, as cuDNN does not return indices information.

Returns When `return_indices` is `False` (default), returns the output variable. When `True`, returns the tuple of the output variable and pooling indices (*N-dimensional array*). Pooling indices will be on the same device as the input.

Return type *Variable* or tuple

chainer.functions.max_pooling_3d

`chainer.functions.max_pooling_3d(x, ksize, stride=None, pad=0, cover_all=True, return_indices=False)`

3-dimensional spatial max pooling function.

Warning: This feature is experimental. The interface can change in the future.

Note: This function calls `max_pooling_nd()` internally, so see the details of the behavior in the documentation of `max_pooling_nd()`.

chainer.functions.max_pooling_nd

`chainer.functions.max_pooling_nd(x, ksize, stride=None, pad=0, cover_all=True, return_indices=False)`

N-dimensionally spatial max pooling function.

Warning: This feature is experimental. The interface can change in the future.

This function provides a N-dimensionally generalized version of `max_pooling_2d()`. This acts similarly to `convolution_nd()`, but it computes the maximum of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

- **x** (`Variable`) – Input variable.
- **ksize** (*int or tuple of ints*) – Size of pooling window. `ksize=k` and `ksize=(k, k, ..., k)` are equivalent.
- **stride** (*int or tuple of ints or None*) – Stride of pooling applications. `stride=s` and `stride=(s,s, ..., s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or tuple of ints*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p, ..., p)` are equivalent.
- **cover_all** (*bool*) – If `True`, all spatial locations are pooled into some output pixels. It may make the output size larger.
- **return_indices** (*bool*) – If `True`, pooling indices array is returned together with the output variable. The returned indices are expected for use by `chainer.functions.upsampling_nd()`. Note that cuDNN will not be used for this function if `return_indices` is set to `True`, as cuDNN does not return indices information.

Returns When `return_indices` is `False` (default), returns the output variable. When `True`, returns the tuple of the output variable and pooling indices (*N-dimensional array*). Pooling indices will be on the same device as the input.

Return type *Variable* or tuple

chainer.functions.roi_average_align_2d

`chainer.functions.roi_average_align_2d`(*x*, *rois*, *roi_indices*, *outsize*, *spatial_scale*, *sampling_ratio=None*)

Spatial Region of Interest (ROI) average align function.

This function acts similarly to `roi_average_pooling_2d()`, but it computes average of input spatial patch with bilinear interpolation for each channel with the region of interest.

Parameters

- **x** (`Variable`) – Input variable. The shape is expected to be 4 dimentional: `(n: batch, c: channel, h, height, w: width)`.
- **rois** (`Variable`) – Input roi variable. The shape is expected to be `(n: data size, 4)`, and each datum is set as below: `(y_min, x_min, y_max, x_max)`.
- **roi_indices** (`Variable`) – Input roi variable. The shape is expected to be `(n: data size, )`.
- **outsize** (*(int, int) or int*) – Expected output size after pooled (height, width). `outsize=o` and `outsize=(o, o)` are equivalent.
- **spatial_scale** (*float*) – Scale of the roi is resized.

- **sampling_ratio** (*(int, int) or int*) – Sampling step for the alignment. It must be an integer over 1 or `None`, and the value is automatically decided when `None` is passed. Use of different ratio in height and width axis is also supported by passing tuple of int as `(sampling_ratio_h, sampling_ratio_w)`. `sampling_ratio=s` and `sampling_ratio=(s, s)` are equivalent.

Returns Output variable.

Return type *Variable*

See the original paper proposing ROIAlign: Mask R-CNN.

chainer.functions.roi_average_pooling_2d

`chainer.functions.`**`roi_average_pooling_2d`**(*x*, *rois*, *roi_indices*, *outsize*, *spatial_scale*)

Spatial Region of Interest (ROI) average pooling function.

This function acts similarly to *average_pooling_2d()*, but it computes the average of input spatial patch for each channel with the region of interest.

Parameters

- **x** (`Variable`) – Input variable. The shape is expected to be 4 dimentional: (n: batch, c: channel, h, height, w: width).
- **rois** (`Variable`) – Input roi variable. The shape is expected to be (n: data size, 4), and each datum is set as below: (y_min, x_min, y_max, x_max).
- **roi_indices** (`Variable`) – Input roi variable. The shape is expected to be (n: data size,).
- **outsize** (*(int, int) or int*) – Expected output size after pooled (height, width). `outsize=o` and `outsize=(o, o)` are equivalent.
- **spatial_scale** (*float*) – Scale of the roi is resized.

Returns Output variable.

Return type *Variable*

See the original paper proposing ROIPooling: Fast R-CNN.

chainer.functions.roi_max_align_2d

`chainer.functions.`**`roi_max_align_2d`**(*x*, *rois*, *roi_indices*, *outsize*, *spatial_scale*, *sampling_ratio=None*)

Spatial Region of Interest (ROI) max align function.

This function acts similarly to *roi_max_pooling_2d()*, but it computes maximum of input spatial patch with bilinear interpolation for each channel with the region of interest.

Parameters

- **x** (`Variable`) – Input variable. The shape is expected to be 4 dimentional: `(n: batch, c: channel, h, height, w: width)`.
- **rois** (`Variable`) – Input roi variable. The shape is expected to be `(n: data size, 4)`, and each datum is set as below: `(y_min, x_min, y_max, x_max)`.
- **roi_indices** (`Variable`) – Input roi variable. The shape is expected to be `(n: data size, )`.

- **outsize** (*(int, int) or int*) – Expected output size after pooled (height, width). `outsize=o` and `outsize=(o, o)` are equivalent.
- **spatial_scale** (*float*) – Scale of the roi is resized.
- **sampling_ratio** (*(int, int) or int*) – Sampling step for the alignment. It must be an integer over 1 or `None`, and the value is automatically decided when `None` is passed. Use of different ratio in height and width axis is also supported by passing tuple of int as `(sampling_ratio_h, sampling_ratio_w)`. `sampling_ratio=s` and `sampling_ratio=(s, s)` are equivalent.

Returns Output variable.

Return type *Variable*

See the original paper proposing ROIAlign: Mask R-CNN.

chainer.functions.roi_max_pooling_2d

`chainer.functions.`**`roi_max_pooling_2d`**(*x*, *rois*, *roi_indices*, *outsize*, *spatial_scale*)

Spatial Region of Interest (ROI) max pooling function.

This function acts similarly to *max_pooling_2d()*, but it computes the maximum of input spatial patch for each channel with the region of interest.

Parameters

- **x** (`Variable`) – Input variable. The shape is expected to be 4 dimentional: (n: batch, c: channel, h, height, w: width).
- **rois** (`Variable`) – Input roi variable. The shape is expected to be (n: data size, 4), and each datum is set as below: (y_min, x_min, y_max, x_max).
- **roi_indices** (`Variable`) – Input roi variable. The shape is expected to be (n: data size,).
- **outsize** (*(int, int) or int*) – Expected output size after pooled (height, width). `outsize=o` and `outsize=(o, o)` are equivalent.
- **spatial_scale** (*float*) – Scale of the roi is resized.

Returns Output variable.

Return type *Variable*

See the original paper proposing ROIPooling: Fast R-CNN.

chainer.functions.roi_pooling_2d

`chainer.functions.`**`roi_pooling_2d`**(*x*, *rois*, *outh*, *outw*, *spatial_scale*)

Spatial Region of Interest (ROI) pooling function.

This function acts similarly to *max_pooling_2d()*, but it computes the maximum of input spatial patch for each channel with the region of interest.

Parameters

- **x** (`Variable`) – Input variable. The shape is expected to be 4 dimentional: (n: batch, c: channel, h, height, w: width).
- **rois** (`Variable`) – Input roi variable. The shape is expected to be (n: data size, 5), and each datum is set as below: (batch_index, x_min, y_min, x_max, y_max).

- **outh** (*int*) – Height of output image after pooled.
- **outw** (*int*) – Width of output image after pooled.
- **spatial_scale** (*float*) – Scale of the roi is resized.

Returns Output variable.

Return type *Variable*

See the original paper proposing ROIPooling: Fast R-CNN.

chainer.functions.spatial_pyramid_pooling_2d

`chainer.functions.spatial_pyramid_pooling_2d`(*x*, *pyramid_height*, *pooling=None*)

Spatial pyramid pooling function.

It outputs a fixed-length vector regardless of input feature map size.

It performs pooling operation to the input 4D-array `x` with different kernel sizes and padding sizes, and then flattens all dimensions except first dimension of all pooling results, and finally concatenates them along second dimension.

At i-th pyramid level, the kernel size $(k_h^{(i)}, k_w^{(i)})$ and padding size $(p_h^{(i)}, p_w^{(i)})$ of pooling operation are calculated as below:

$$\begin{aligned} k_h^{(i)} &= \lceil b_h / 2^i \rceil, \\ k_w^{(i)} &= \lceil b_w / 2^i \rceil, \\ p_h^{(i)} &= (2^i k_h^{(i)} - b_h)/2, \\ p_w^{(i)} &= (2^i k_w^{(i)} - b_w)/2, \end{aligned}$$

where $\lceil \cdot \rceil$ denotes the ceiling function, and b_h, b_w are height and width of input variable `x`, respectively. Note that index of pyramid level i is zero-based.

See detail in paper: Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition.

Parameters

- **x** (`Variable`) – Input variable. The shape of `x` should be `(batchsize, # of channels, height, width)`.
- **pyramid_height** (*int*) – Number of pyramid levels
- **pooling** (*str*) – Currently, only `max` is supported, which performs a 2d max pooling operation.

Returns Output variable. The shape of the output variable will be $(batchsize, c \sum_{h=0}^{H-1} 2^{2h}, 1, 1)$, where c is the number of channels of input variable `x` and H is the number of pyramid levels.

Return type *Variable*

chainer.functions.unpooling_1d

`chainer.functions.unpooling_1d`(*x*, *ksize*, *stride=None*, *pad=0*, *outsize=None*, *cover_all=True*)

Inverse operation of 1-dimensional spatial pooling.

> **Warning:** This feature is experimental. The interface can change in the future.

Note: This function calls `unpooling_nd()` internally, so see the details of the behavior in the documentation of `unpooling_nd()`.

chainer.functions.unpooling_2d

`chainer.functions.unpooling_2d`(*x*, *ksize*, *stride=None*, *pad=0*, *outsize=None*, *cover_all=True*)

Inverse operation of pooling for 2d array.

This function acts similarly to `Deconvolution2DFunction`, but it spreads input 2d array's value without any parameter instead of computing the inner products.

Parameters

- **x** (`Variable`) – Input variable.
- **ksize** (*int or pair of ints*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int, pair of ints or None*) – Stride of pooling applications. `stride=s` and `stride=(s, s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or pair of ints*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p)` are equivalent.
- **outsize** (*None or pair of ints*) – Expected output size (height, width) of array after the operation. If `None`, the size (height or width) is estimated from the size of input array in first batch with `get_deconv_outsize()`. If outsize is not `None`, the result of outsize applied to `get_conv_outsize()` must be equal to the shape of the 2d array in the input batch `x`.
- **cover_all** (*bool*) – If `True`, the output size may be smaller than the size if `cover_all` is `False`. This flag serves to align behavior to the pooling functions which can cover all input locations, see `max_pooling_2d()` and `convolution_2d()`.

Returns Output variable.

Return type *Variable*

chainer.functions.unpooling_3d

`chainer.functions.unpooling_3d`(*x*, *ksize*, *stride=None*, *pad=0*, *outsize=None*, *cover_all=True*)

Inverse operation of 3-dimensional spatial pooling.

Warning: This feature is experimental. The interface can change in the future.

Note: This function calls `unpooling_nd()` internally, so see the details of the behavior in the documentation of `unpooling_nd()`.

chainer.functions.unpooling_nd

`chainer.functions.unpooling_nd(x, ksize, stride=None, pad=0, outsize=None, cover_all=True)`

Inverse operation of N-dimensional spatial pooling.

> **Warning:** This feature is experimental. The interface can change in the future.

This function acts similarly to `DeconvolutionND`, but it spreads input N-dimensional array's value without any parameter instead of computing the inner products.

Parameters

- **x** (`Variable`) – Input variable.
- **ksize** (*int or pair of ints*) – Size of pooling window $(k_1, k_2, ..., k_N)$. `ksize=k` is equivalent to `(k, k, ..., k)`.
- **stride** (*int, pair of ints or None*) – Stride of pooling applications $(s_1, s_2, ..., s_N)$. `stride=s` is equivalent to `(s, s, ..., s)`. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*int or pair of ints*) – Spatial padding width for the input array $(p_1, p_2, ..., p_N)$. `pad=p` is equivalent to `(p, p, ..., p)`.
- **outsize** (*None or pair of ints*) – Expected output size of unpooling operation $(out_1, out_2, ..., out_N)$. If `None`, the size is estimated from input size, stride and padding.
- **cover_all** (*bool*) – If `True`, the pooling window is assumed to cover all of the output array, eventually the output size may be smaller than that in the case `cover_all` is `False`.

Returns Output variable.

Return type *Variable*

chainer.functions.upsampling_2d

`chainer.functions.upsampling_2d(x, indexes, ksize, stride=None, pad=0, outsize=None, cover_all=True)`

Upsampling using pooling indices.

This function produces an upsampled image using pooling indices.

Example

```
>>> x = np.arange(1, 37).reshape(1, 1, 6, 6).astype(np.float32)
>>> x = chainer.Variable(x)
>>> x.array
array([[[[ 1.,  2.,  3.,  4.,  5.,  6.],
         [ 7.,  8.,  9., 10., 11., 12.],
         [13., 14., 15., 16., 17., 18.],
         [19., 20., 21., 22., 23., 24.],
         [25., 26., 27., 28., 29., 30.],
         [31., 32., 33., 34., 35., 36.]]]], dtype=float32)
```

This is the original `x` before max pooling.

```
>>> pooled_x, indexes = F.max_pooling_2d(
...     x, ksize=2, stride=2, return_indices=True)
>>> pooled_x.array
array([[[[ 8., 10., 12.],
         [20., 22., 24.],
         [32., 34., 36.]]]], dtype=float32)
>>> indexes
array([[[[3, 3, 3],
         [3, 3, 3],
         [3, 3, 3]]]])
```

These are the outputs from the max pooling operation including the resulting indices that will be used to upsample `pooled_x`. Note that the indices all point to the largest, in the case the last, elements in each window.

```
>>> upsampled_x = F.upsampling_2d(
...     pooled_x, indexes, ksize=2, stride=2, outsize=x.shape[2:])
>>> upsampled_x.shape
(1, 1, 6, 6)
>>> upsampled_x.array
array([[[[ 0.,  0.,  0.,  0.,  0.,  0.],
         [ 0.,  8.,  0., 10.,  0., 12.],
         [ 0.,  0.,  0.,  0.,  0.,  0.],
         [ 0., 20.,  0., 22.,  0., 24.],
         [ 0.,  0.,  0.,  0.,  0.,  0.],
         [ 0., 32.,  0., 34.,  0., 36.]]]], dtype=float32)
```

Parameters

- **x** (`Variable`) – Input variable.
- **indexes** (*N-dimensional array*) – Index array returned from preceding call to *`max_pooling_2d()`*.
- **ksize** (*`int or pair of ints`*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*`int or pair of ints or None`*) – Stride of pooling applications. `stride=s` and `stride=(s, s)` are equivalent. If `None` is specified, then it uses same stride as the pooling window size.
- **pad** (*`int or pair of ints`*) – Spatial padding width for the input array. `pad=p` and `pad=(p, p)` are equivalent.
- **outsize** (*`(int, int)`*) – Expected output size (height, width).
- **cover_all** (*`bool`*) – Should be set to `True` if all spatial locations were pooled into some output pixels during the preceding pooling operation. `False` otherwise. See *`max_pooling_2d()`*.

Returns Output variable.

Return type *Variable*

4.2.11 Utility functions

chainer.functions.forget	Calls a function without storing intermediate results.

chainer.functions.forget

`chainer.functions.forget(func, *xs)`

Calls a function without storing intermediate results.

On a forward propagation, Chainer normally stores all intermediate results of *VariableNode*s on a computational graph as they are required on backward propagation. Sometimes these results consume too much memory. `F.forget` *forgets* such intermediate results on forward propagation, and still supports backpropagation with recalculation.

On a forward propagation, `F.forget` calls a given function with given variables without creating a computational graph. That means, no intermediate results are stored. On a backward propagation, `F.forget` calls the given function again to create a computational graph for backpropagation.

`F.forget` reduces internal memory usage, whereas it requires more calculation time as it calls the function twice.

Example

Let `f` be a function defined as:

```
>>> def f(a, b):
...   return (a + b) * a
```

and, `x` and `y` be *Variable*s:

```
>>> x = chainer.Variable(np.random.uniform(-1, 1, 5).astype(np.float32))
>>> y = chainer.Variable(np.random.uniform(-1, 1, 5).astype(np.float32))
```

When `z` is calculated as `z = f(x, y)`, its intermediate result `x + y` is stored in memory. Instead, if you call `f` with `F.forget`:

```
>>> z = F.forget(f, x, y)
```

intermediate `x + y` is forgotten.

Note: `F.forget` does not support functions which behave differently in multiple calls with the same inputs, such as *F.dropout()* and *F.negative_sampling()*.

Note: In case input argument variables are of *N-dimensional array* objects, arguments will automatically be converted to *Variable*s. This conversion takes place to ensure that this function is included in the computational graph to enable backward computations.

Note: `F.forget` does not support double backpropagation.

Note: If you want to use `F.forget` to a link which updates the link's internal information every time the forward computation is called, please ensure that the information is updated just once in a single iteration. You may use the `chainer.config.in_recomputing` flag to check if the forward computation is the first call in an iteration. Please see the implementation of `BatchNormalization` for detail.

Parameters

- **func** (*callable*) – A function to call. It needs to be called with `Variable` object(s) and to return a `Variable` object or a tuple of `Variable` objects.
- **xs** (`tuple` of `Variable` or *N-dimensional array*) – Argument variables of the function.

Returns A variable `func` returns. If it returns a tuple, the method returns a tuple too.

Return type *Variable*

4.2.12 Function base

`chainer.Function`	Old-style interface of a differentiable function.
`chainer.FunctionAdapter`	Adapter class to wrap Function with FunctionNode.
`chainer.FunctionNode`	Function node of the computational graph.
`chainer.force_backprop_mode`	Make a context manager which enables back-propagation.
`chainer.no_backprop_mode`	Make a context manager which disables back-propagation.
`chainer.grad`	Computes the gradient of output variables w.r.t. the input variables.

chainer.Function

class chainer.**Function**

Old-style interface of a differentiable function.

This class provides an interface to implement an old-style differentiable function (i.e., the function application is recorded to the computational graph). The subclass of `Function` that implement `forward()` and `backward()` can be used to run the forward computation and automatically induce the backpropagation procedure.

There is another way to implement such a function: subclassing `FunctionNode`. There are mainly two differences between them.

1. The *differentiable backprop* is available for `FunctionNode`, while it is not for `Function` because the `backward()` of the latter directly operates on the arrays instead of `Variable` objects so that it cannot record the history of the computation.
2. The information passed to `backward()` is different. In `FunctionNode`, which inputs the function node has to compute the gradients w.r.t. is passed so that it can omit unnecessary computations, while `Function` always has to compute gradients w.r.t. all the input nodes. The `FunctionNode` also accepts the current gradient values of the input nodes so that the accumulation work can be merged with the gradient computation if an efficient kernel is available.

This class uses `FunctionAdapter` to convert the interface to that of `FunctionNode` and adds the `FunctionNode` object to the computational graph.

See `FunctionNode` for the details of building the computational graph in Chainer.

Methods

`__call__`(**inputs*)
Applies forward propagation with chaining backward references.

This method creates a new `FunctionAdapter` object and runs the forward propagation using it.

See `FunctionNode` for the detailed behavior of building the computational graph.

Parameters **`inputs`** – Tuple of input `Variable` or *N-dimensional array* objects. If the input is *N-dimensional array*, it is automatically wrapped with `Variable`.

Returns One `Variable` object or a tuple of multiple `Variable` objects.

`add_hook`(*hook*, *name=None*)
Registers a function hook.

See `FunctionNode.add_hook()` for the detail.

Parameters

- **`hook`** (`FunctionHook`) – Function hook to be registered.
- **`name`** (`str`) – Name of the function hook. name must be unique among function hooks registered to the function. If `None`, default name of the function hook is used.

`backward`(*inputs*, *grad_outputs*)
Applies backprop to output gradient arrays.

It delegates the procedure to `backward_cpu()` or `backward_gpu()` by default. Which it selects is determined by the type of input arrays and output gradient arrays. Implementations of `Function` must implement either CPU/GPU methods or this method, if the function is intended to be backprop-ed.

Parameters

- **`inputs`** – Tuple of input arrays.
- **`grad_outputs`** – Tuple of output gradient arrays.

Returns Tuple of input gradient arrays. Some or all of them can be `None`, if the function is not differentiable on inputs.

Return type tuple

> **Warning:** Implementations of `Function` must take care that the return value must be a tuple even if it returns only one array.

`backward_cpu`(*inputs*, *grad_outputs*)
Applies backprop to output gradient arrays on CPU.

Parameters

- **`inputs`** – Tuple of input `numpy.ndarray` object(s).
- **`grad_outputs`** – Tuple of output gradient `numpy.ndarray` object(s).

Returns Tuple of input gradient `numpy.ndarray` object(s). Some or all of them can be `None`, if the function is not differentiable on corresponding inputs.

Return type tuple

> **Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

backward_gpu (*inputs*, *grad_outputs*)

Applies backprop to output gradient arrays on GPU.

Parameters

- **inputs** – Tuple of input cupy.ndarray object(s).
- **grad_outputs** – Tuple of output gradient cupy.ndarray object(s).

Returns Tuple of input gradient cupy.ndarray object(s). Some or all of them can be None, if the function is not differentiable on corresponding inputs.

Return type tuple

> **Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

check_type_forward (*in_types*)

Checks types of input data before forward propagation.

Before *forward()* is called, this function is called. You need to validate types of input data in this function using *the type checking utilities*.

Parameters in_types (TypeInfoTuple) – The type information of input data for *forward()*.

delete_hook (*name*)

Unregisters the specified function hook.

Parameters name (*str*) – the name of the function hook to be unregistered.

forward (*inputs*)

Applies forward propagation to input arrays.

It delegates the procedure to *forward_cpu()* or *forward_gpu()* by default. Which it selects is determined by the type of input arrays. Implementations of *Function* must implement either CPU/GPU methods or this method.

Parameters inputs – Tuple of input array(s).

Returns Tuple of output array(s).

> **Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

forward_cpu (*inputs*)

Applies forward propagation to input arrays on CPU.

Parameters inputs – Tuple of numpy.ndarray object(s).

Returns Tuple of numpy.ndarray object(s).

Return type tuple

> **Warning:** Implementations of `Function` must take care that the return value must be a tuple even if it returns only one array.

forward_gpu(*inputs*)

Applies forward propagation to input arrays on GPU.

Parameters **inputs** – Tuple of `cupy.ndarray` object(s).

Returns Tuple of `cupy.ndarray` object(s).

Return type tuple

> **Warning:** Implementations of `Function` must take care that the return value must be a tuple even if it returns only one array.

retain_inputs(*indexes*)

Lets specified input variable nodes keep data arrays.

By calling this method from `forward()`, the function can specify which inputs are required for backprop.

If this method is not called, the function keeps all input arrays. If you want to release all input arrays, call this method by passing an empty sequence. *Note that this behavior is different from that of* `FunctionNode.retain_inputs()`.

Note that **this method must not be called from the outside of** `forward()`.

Parameters **indexes** (*iterable of int*) – Indexes of input variables that the function will require for backprop.

retain_outputs(*indexes*, *retain_after_backward=False*)

Lets specified output variable nodes keep data arrays.

By calling this method from `forward()`, the function can specify which outputs are required for backprop. If this method is not called, any output variables are not marked to keep the data array at the point of returning from `__call__()`. The retained arrays are stored to `output_data`.

> **Note:** It is STRONGLY RECOMMENDED that you use this method if the function requires some or all output arrays in backprop. The function can also use output arrays just by keeping references to them directly, whereas it might influence on the performance of later function applications to the output variables.

Note that **this method must not be called from the outside of** `forward()`.

Parameters

- **indexes** (*iterable of int*) – Indexes of input variables that the function will require for backprop.
- **retain_after_backward** (*bool*) – This option has no effect. It is left only for the backward compatibility.

unchain()

Purges in/out nodes and this function itself from the graph.

See `FunctionNode.unchain()` for the detail.

__eq__()

Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

inputs
The input nodes of the function.

label
Short text that represents the function.

The default implementation returns its type name. Each function should override it to give more information.

local_function_hooks
Ordered Dictionary of registered function hooks.

See *FunctionNode.local_function_hooks* for the detail.

node
The *FunctionAdapter* object that wraps this Function.

If the Function does not have a node object, this property automatically creates a new one.

output_data
A tuple of the retained output arrays.

It has the same length as the *outputs*. Elements that are not retained are set to `None`.

outputs
Weak references to the output nodes of the function.

rank
The topological ordinal of the corresponding function node.

stack

chainer.FunctionAdapter

class chainer.**FunctionAdapter**(*function*)
Adapter class to wrap Function with FunctionNode.

While *FunctionNode* provides the interface of new-style differentiable functions, the old-style *Function* can still be used for the backward compatibility. This class provides an adapter of there interface; it adds *FunctionNode* interface to any *Function* object by delegation.

Note: The ownership of *FunctionAdapter* and *Function* is a bit tricky. At the initialization, *FunctionAdapter* is owned by the *Function* object. Once the function is applied to variables, the ownership is reversed; the adapter becomes the owner of the *Function* object and the *Function* object changes the reference to a weak one.

Parameters **function** (Function) – The function object to wrap.

New in version 3.0.0.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a function hook.

Parameters

- **hook** (FunctionHook) – Function hook to be registered.
- **name** (*str*) – Name of the function hook. The name must be unique among function hooks registered to this function. If `None`, the default name of the function hook is used.

apply(*inputs*)
Computes output variables and grows the computational graph.

Basic behavior is expressed in the documentation of *FunctionNode*.

Note: If the *data* attributes of the input variables exist on a GPU device, that device is made current before calling *forward()*, so implementers do not need to take care of device selection in most cases.

Parameters **inputs** – Tuple of input variables. Each element can be either *Variable* or *N-dimensional array*. If the element is an ndarray, it is automatically wrapped with *Variable*.

Returns A tuple of output *Variable* objects.

backward(*target_input_indexes*, *grad_outputs*)
Computes gradients w.r.t. specified inputs given output gradients.

This method is used to compute one step of the backpropagation corresponding to the forward computation of this function node. Given the gradients w.r.t. output variables, this method computes the gradients w.r.t. specified input variables. Note that this method does not need to compute any input gradients not specified by `target_input_indices`.

Unlike *Function.backward()*, gradients are given as *Variable* objects and this method itself has to return input gradients as *Variable* objects. It enables the function node to return the input gradients with the full computational history, in which case it supports *differentiable backpropagation* or *higher-order differentiation*.

The default implementation returns `None` s, which means the function is not differentiable.

Parameters

- **target_input_indexes** (*tuple of int*) – Sorted indices of the input variables w.r.t. which the gradients are required. It is guaranteed that this tuple contains at least one element.
- **grad_outputs** (tuple of *Variables*) – Gradients w.r.t. the output variables. If the gradient w.r.t. an output variable is not given, the corresponding element is `None`.

Returns Tuple of variables that represent the gradients w.r.t. specified input variables. The length of the tuple can be same as either `len(target_input_indexes)` or the number of inputs. In the latter case, the elements not specified by `target_input_indexes` will be discarded.

See also:

backward_accumulate() provides an alternative interface that allows you to implement the backward computation fused with the gradient accumulation.

backward_accumulate(*target_input_indexes*, *grad_outputs*, *grad_inputs*)
Computes gradients w.r.t. specified inputs and accumulates them.

This method provides a way to fuse the backward computation and the gradient accumulations in the case that the multiple functions are applied to the same variable.

Users have to override either of this method or *backward()*. It is often simpler to implement *backward()* and is recommended if you do not need to provide efficient gradient accumulation.

Parameters

- **target_input_indexes** (*tuple of int*) – Sorted indices of the input variables w.r.t. which the gradients are required. It is guaranteed that this tuple contains at least one element.
- **grad_outputs** (*tuple of Variable*) – Gradients w.r.t. the output variables. If the gradient w.r.t. an output variable is not given, the corresponding element is `None`.
- **grad_inputs** (*tuple of Variable*) – Gradients w.r.t. the input variables specified by `target_input_indexes`. These values are computed by other computation paths. If there is no gradient value existing for the variable, the corresponding element is `None`. See also the note below.

Returns Tuple of variables that represent the gradients w.r.t. specified input variables. Unlike *backward()*, the length of the tuple **must** be same as that of `target_input_indices`.

Note: Gradient variables in `grad_outputs` are distinct, even if a variable is passed to multiple input arguments of the function. This is an implementation-detail convention to avoid the complication of correctly accumulating gradients in such a case.

Usually, only the first position of `grad_inputs` corresponding to these input arguments may contain the gradient variable corresponding to that input variable, and other entries are set to `None`. This is not the case with the `lazy_grad_sum` feature. This behavior might be changed in a future version.

---

check_type_forward(*in_types*)
Checks types of input data before forward propagation.

This method is called before *forward()* and validates the types of input variables using *the type checking utilities*.

Parameters **in_types** (`TypeInfoTuple`) – The type information of input variables for *forward()*.

delete_hook(*name*)
Unregisters the function hook.

Parameters **name** (*str*) – The name of the function hook to be unregistered.

forward(*inputs*)
Computes the output arrays from the input arrays.

It delegates the procedure to `forward_cpu()` or `forward_gpu()` by default. Which of them this method selects is determined by the type of input arrays. Implementations of `FunctionNode` must implement either CPU/GPU methods or this method.

Parameters **inputs** – Tuple of input array(s).

Returns Tuple of output array(s).

> **Warning:** Implementations of `FunctionNode` must take care that the return value must be a tuple even if it returns only one array.

forward_chainerx(*inputs*)
Computes the output arrays from the input ChainerX arrays.

This method may check the input arrays and other attributes to see if the computation can be done using ChainerX implementation. If it's not supported, `chainer.Fallback` should be returned instead of output arrays. In that case, computation using conventional Python implementation will be performed.

Parameters **inputs** – Tuple of input array(s).

Returns Tuple of output array(s) or `chainer.Fallback`.

forward_cpu(*inputs*)
Computes the output arrays from the input NumPy arrays.

Parameters **inputs** – Tuple of input `numpy.ndarray` objects.

Returns Tuple of output arrays. Each element can be NumPy or CuPy arrays.

> **Warning:** Implementation of `FunctionNode` must take care that the return value must be a tuple even if it returns only one array.

forward_gpu(*inputs*)
Computes the output arrays from the input CuPy arrays.

Parameters **inputs** – Tuple of input `cupy.ndarray` objects.

Returns Tuple of output arrays. Each element can be NumPy or CuPy arrays.

> **Warning:** Implementation of `FunctionNode` must take care that the return value must be a tuple even if it returns only one array.

get_retained_inputs()
Returns a tuple of retained input variables.

This method is used to retrieve the input variables retained in `forward()`.

Returns A tuple of retained input variables, if available. Otherwise return *None*.

get_retained_outputs()
Returns a tuple of retained output variables.

This method is used to retrieve the output variables retained in `forward()`.

Returns A tuple of retained output variables, if available. Otherwise return *None*.

Note: This method does a tricky thing to support the case of an output node garbage-collected before this method is called; in this case, this method creates a fresh variable node that acts as an output node of the function node.

---

retain_inputs(*indexes*)
Lets specified input variable nodes keep data arrays.

By calling this method from `forward()`, the function node can specify which inputs are required for backprop. The input variables with retained arrays can then be obtained by calling `get_retained_inputs()` from inside `backward()`.

Unlike `Function`, the function node **DOES NOT** keep input arrays by default. If you want to keep some or all input arrays, do not forget to call this method.

Note that **this method must not be called from the outside of** `forward()`.

Parameters **indexes** (*iterable of int*) – Indexes of input variables that the function will require for backprop.

retain_outputs(*indexes*)
Lets specified output variable nodes keep data arrays.

By calling this method from `forward()`, the function node can specify which outputs are required for backprop. If this method is not called, no output variables will be marked to keep their data array at the point of returning from `apply()`. The output variables with retained arrays can then be obtained by calling `get_retained_outputs()` from inside `backward()`.

Note: It is recommended to use this method if the function requires some or all output arrays in backprop. The function can also use output arrays just by keeping references to them directly, although it might affect the performance of later function applications on the output variables.

Note that **this method must not be called from the outside of** `forward()`.

Parameters **indexes** (*iterable of int*) – Indexes of output variables that the function will require for backprop.

unchain()
Purges in/out nodes and this function node itself from the graph.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

chainerx_device = None

function
The *Function* object that this adapter is wrapping.

inputs = None

label
Short text that represents the function.

The default implementation returns its type name. Each function should override it to give more information.

lazy_grad_sum = False

local_function_hooks
Ordered dictionary of registered function hooks.

Contrary to `chainer.thread_local.function_hooks`, which registers its elements to all functions, Function hooks in this property is specific to this function.

output_data
A tuple of the retained output arrays.

This property is mainly used by *Function*. Users basically do not have to use this property; use *get_retained_outputs()* instead.

outputs = None

rank = 0

stack = None

chainer.FunctionNode

class chainer.**FunctionNode**
Function node of the computational graph.

FunctionNode is a class representing a node in a computational graph. The node corresponds to an application of a differentiable function to input variables.

When a differentiable function is applied to *Variable* objects, it creates an instance of FunctionNode implementation and calls its *apply()* method. The *apply()* method basically does the following three things.

1. Adding an edge from the function node to the variable node corresponding to each input. The node of each input is extracted by *Variable.node*.
2. Computing the output arrays of the function.
3. Creating a *Variable* object for each output array and adding an edge from the node of the variable to the function node.

The output variables are then returned.

Example

Let `x` be an instance of *Variable* and `f` be an instance of *FunctionNode* taking only one argument. Then the following code

```
>>> import numpy, chainer
>>> x = chainer.Variable(numpy.zeros(10))
>>> f = chainer.functions.math.identity.Identity()
>>> y = f.apply((x,))[0]
```

computes a new variable `y` and creates backward references. The backward references are actually set as per the following diagram:

```
x.node <--- f <--- y.node
```

If an application of another function `g` occurs as

```
>>> g = chainer.functions.math.identity.Identity()
>>> z = g.apply((x,))[0]
```

then the graph grows with a branch:

```
         |--- f <--- y.node
x.node <-+
         |--- g <--- z.node
```

Note that the branching is correctly managed on backward computation, i.e. the gradients from `f` and `g` are accumulated to the gradient of `x`.

Every function-node implementation should provide *forward()* and *backward()*. Instead of overriding *forward()*, one can also implement *forward_cpu()* and *forward_gpu()* when the implementations for CPU and GPU arrays are totally different.

Note that the input and output variables are inaccessible from *backward()* by default. If it needs accesses to these variables, the *forward()* method (or its CPU/GPU variants) has to call *retain_inputs()* and *retain_outputs()* appropriately. The retained input/output variables can be accessed from *backward()* by calling *get_retained_inputs()* and *get_retained_outputs()*.

Note: There are two types of differentiable functions in Chainer (since v3). The first type is of a function using a subclass of *Function*, which is called *old-style differentiable function*. The second type is of a function using a subclass of *FunctionNode*, which is called **new-style differentiable function**. There are several advantages on using the new-style differentiable function.

- The new-style differentiable function supports *differentiable backpropagation*. The backpropagated gradients computed through the new-style differentiable functions themselves support further backpropagations so that the automatic higher-order differentiation is available.
- The backpropagation of the new-style differentiable function can be more computationally efficient because the interface allows an implementation to omit the computation of unneeded input gradients.

Note that the new-style differentiable function is the standard way of defining a function node of the computational graph in Chainer; old- style differentiable functions are implemented as wrappers of the new- style differentiable functions.

Variables

- ***inputs*** – A tuple of the input *VariableNode* objects.
- ***outputs*** – A tuple of weak references to the output *VariableNode* objects.
- ***rank*** (*int*) – An ordinal following the topological order of the computational graph.
- ***stack*** – Stack trace retrieved at the forward computation. The stack trace is available only in the debug mode.

New in version 3.0.0.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a function hook.

Parameters

- **hook** (`FunctionHook`) – Function hook to be registered.
- **name** (*str*) – Name of the function hook. The name must be unique among function hooks registered to this function. If `None`, the default name of the function hook is used.

`apply`(*inputs*)
Computes output variables and grows the computational graph.

Basic behavior is expressed in the documentation of *FunctionNode*.

Note: If the *data* attributes of the input variables exist on a GPU device, that device is made current before calling *forward()*, so implementers do not need to take care of device selection in most cases.

Parameters inputs – Tuple of input variables. Each element can be either *Variable* or *N-dimensional array*. If the element is an ndarray, it is automatically wrapped with *Variable*.

Returns A tuple of output *Variable* objects.

`backward`(*target_input_indexes*, *grad_outputs*)
Computes gradients w.r.t. specified inputs given output gradients.

This method is used to compute one step of the backpropagation corresponding to the forward computation of this function node. Given the gradients w.r.t. output variables, this method computes the gradients w.r.t. specified input variables. Note that this method does not need to compute any input gradients not specified by `target_input_indices`.

Unlike *Function.backward()*, gradients are given as *Variable* objects and this method itself has to return input gradients as *Variable* objects. It enables the function node to return the input gradients with the full computational history, in which case it supports *differentiable backpropagation* or *higher-order differentiation*.

The default implementation returns `None` s, which means the function is not differentiable.

Parameters

- **target_input_indexes** (*tuple of int*) – Sorted indices of the input variables w.r.t. which the gradients are required. It is guaranteed that this tuple contains at least one element.
- **grad_outputs** (tuple of *Variable*s) – Gradients w.r.t. the output variables. If the gradient w.r.t. an output variable is not given, the corresponding element is `None`.

Returns Tuple of variables that represent the gradients w.r.t. specified input variables. The length of the tuple can be same as either `len(target_input_indexes)` or the number of inputs. In the latter case, the elements not specified by `target_input_indexes` will be discarded.

See also:

backward_accumulate() provides an alternative interface that allows you to implement the backward computation fused with the gradient accumulation.

backward_accumulate(*target_input_indexes*, *grad_outputs*, *grad_inputs*)

Computes gradients w.r.t. specified inputs and accumulates them.

This method provides a way to fuse the backward computation and the gradient accumulations in the case that the multiple functions are applied to the same variable.

Users have to override either of this method or *backward()*. It is often simpler to implement *backward()* and is recommended if you do not need to provide efficient gradient accumulation.

Parameters

- **target_input_indexes** (*tuple of int*) – Sorted indices of the input variables w.r.t. which the gradients are required. It is guaranteed that this tuple contains at least one element.
- **grad_outputs** (*tuple of Variable*) – Gradients w.r.t. the output variables. If the gradient w.r.t. an output variable is not given, the corresponding element is `None`.
- **grad_inputs** (*tuple of Variable*) – Gradients w.r.t. the input variables specified by `target_input_indexes`. These values are computed by other computation paths. If there is no gradient value existing for the variable, the corresponding element is `None`. See also the note below.

Returns Tuple of variables that represent the gradients w.r.t. specified input variables. Unlike *backward()*, the length of the tuple **must** be same as that of `target_input_indices`.

Note: Gradient variables in `grad_outputs` are distinct, even if a variable is passed to multiple input arguments of the function. This is an implementation-detail convention to avoid the complication of correctly accumulating gradients in such a case.

Usually, only the first position of `grad_inputs` corresponding to these input arguments may contain the gradient variable corresponding to that input variable, and other entries are set to `None`. This is not the case with the `lazy_grad_sum` feature. This behavior might be changed in a future version.

---

check_type_forward(*in_types*)

Checks types of input data before forward propagation.

This method is called before *forward()* and validates the types of input variables using *the type checking utilities*.

Parameters **in_types** (`TypeInfoTuple`) – The type information of input variables for *forward()*.

delete_hook(*name*)
Unregisters the function hook.

Parameters **name** (*str*) – The name of the function hook to be unregistered.

forward(*inputs*)
Computes the output arrays from the input arrays.

It delegates the procedure to *forward_cpu()* or *forward_gpu()* by default. Which of them this method selects is determined by the type of input arrays. Implementations of *FunctionNode* must implement either CPU/GPU methods or this method.

Parameters **inputs** – Tuple of input array(s).

Returns Tuple of output array(s).

> **Warning:** Implementations of *FunctionNode* must take care that the return value must be a tuple even if it returns only one array.

forward_chainerx(*inputs*)
Computes the output arrays from the input ChainerX arrays.

This method may check the input arrays and other attributes to see if the computation can be done using ChainerX implementation. If it's not supported, `chainer.Fallback` should be returned instead of output arrays. In that case, computation using conventional Python implementation will be performed.

Parameters **inputs** – Tuple of input array(s).

Returns Tuple of output array(s) or `chainer.Fallback`.

forward_cpu(*inputs*)
Computes the output arrays from the input NumPy arrays.

Parameters **inputs** – Tuple of input `numpy.ndarray` objects.

Returns Tuple of output arrays. Each element can be NumPy or CuPy arrays.

> **Warning:** Implementation of *FunctionNode* must take care that the return value must be a tuple even if it returns only one array.

forward_gpu(*inputs*)
Computes the output arrays from the input CuPy arrays.

Parameters **inputs** – Tuple of input `cupy.ndarray` objects.

Returns Tuple of output arrays. Each element can be NumPy or CuPy arrays.

> **Warning:** Implementation of *FunctionNode* must take care that the return value must be a tuple even if it returns only one array.

get_retained_inputs()
Returns a tuple of retained input variables.

This method is used to retrieve the input variables retained in *forward()*.

Returns A tuple of retained input variables, if available. Otherwise return *None*.

get_retained_outputs()
Returns a tuple of retained output variables.

This method is used to retrieve the output variables retained in `forward()`.

Returns A tuple of retained output variables, if available. Otherwise return *None*.

Note: This method does a tricky thing to support the case of an output node garbage-collected before this method is called; in this case, this method creates a fresh variable node that acts as an output node of the function node.

---

retain_inputs(*indexes*)
Lets specified input variable nodes keep data arrays.

By calling this method from `forward()`, the function node can specify which inputs are required for backprop. The input variables with retained arrays can then be obtained by calling `get_retained_inputs()` from inside `backward()`.

Unlike `Function`, the function node **DOES NOT** keep input arrays by default. If you want to keep some or all input arrays, do not forget to call this method.

Note that **this method must not be called from the outside of** `forward()`.

Parameters **indexes** (*iterable of int*) – Indexes of input variables that the function will require for backprop.

retain_outputs(*indexes*)
Lets specified output variable nodes keep data arrays.

By calling this method from `forward()`, the function node can specify which outputs are required for backprop. If this method is not called, no output variables will be marked to keep their data array at the point of returning from `apply()`. The output variables with retained arrays can then be obtained by calling `get_retained_outputs()` from inside `backward()`.

Note: It is recommended to use this method if the function requires some or all output arrays in backprop. The function can also use output arrays just by keeping references to them directly, although it might affect the performance of later function applications on the output variables.

Note that **this method must not be called from the outside of** `forward()`.

Parameters **indexes** (*iterable of int*) – Indexes of output variables that the function will require for backprop.

unchain()
Purges in/out nodes and this function node itself from the graph.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

chainerx_device = None

inputs = None

label
Short text that represents the function.

The default implementation returns its type name. Each function should override it to give more information.

lazy_grad_sum = False

local_function_hooks
Ordered dictionary of registered function hooks.

Contrary to `chainer.thread_local.function_hooks`, which registers its elements to all functions, Function hooks in this property is specific to this function.

output_data
A tuple of the retained output arrays.

This property is mainly used by *Function*. Users basically do not have to use this property; use *get_retained_outputs()* instead.

outputs = None

rank = 0

stack = None

chainer.force_backprop_mode

chainer.**force_backprop_mode**()
Make a context manager which enables back-propagation.

When you want to enable back-propagation in *no_backprop_mode()*, call this method. A *Variable* created in this context always has a computational graph unless overridden by deeper contexts. If you call this method outside of *no_backprop_mode()* context, it changes nothing.

In the following example, `y` has a computational graph and calling *backward()* on `y` will compute and accumulate the gradients of the variables in the graph, in this case only `x`.

```
>>> x = chainer.Variable(np.array([1,], np.float32))
>>> with chainer.no_backprop_mode():
...     with chainer.force_backprop_mode():
...         y = x + 1
>>> y.backward()
>>> x.grad
array([1.], dtype=float32)
```

Note: `chainer.force_backprop_mode()` implicitly applies ChainerX's counterpart `chainerx.force_backprop_mode()`, but not vice versa. Also, setting `enable_backprop` *configuration* does not affect ChainerX.

See also:

See `chainer.no_backprop_mode()` for details on disabled back-propagation mode.

chainer.no_backprop_mode

`chainer.`**`no_backprop_mode`**`()`

Make a context manager which disables back-propagation.

In this context, Chainer does not make a computational graph. It has the benefit of reducing memory consumption. However, a `Variable` created in this context does not hold a reference to the `FunctionNode` that created itself so no gradients are accumulated by `backward()`.

In the following example, `y` is created in this context, which means that calling `backward()` on `y` has no effect on the gradients of `x`.

```
>>> x = chainer.Variable(np.array([1,], np.float32))
>>> with chainer.no_backprop_mode():
...     y = x + 1
>>> y.backward()
>>> x.grad is None
True
```

Note: `chainer.no_backprop_mode()` implicitly applies ChainerX's counterpart `chainerx.no_backprop_mode()`, but not vice versa. Also, setting `enable_backprop` *configuration* does not affect ChainerX.

See also:

See `chainer.force_backprop_mode()` for details on how to override this context.

chainer.grad

`chainer.`**`grad`**`(`*outputs*, *inputs*, *grad_outputs=None*, *grad_inputs=None*, *set_grad=False*, *retain_grad=False*, *enable_double_backprop=False*, *loss_scale=None*`)`

Computes the gradient of output variables w.r.t. the input variables.

This function implements the backpropagation algorithm. While `Variable.backward()` also implements backprop, this function selects the smallest paths in the computational graph needed to compute the gradients w.r.t. inputs. The error is backpropagated only through these selected paths, which may reduce the overall computational cost.

This function also differs from `Variable.backward()` in the way to return the gradients; it directly returns the gradient variables as a list instead of setting gradients to the `Variable.grad_var` attribute of the original variable. It means users do not need to clear the gradient w.r.t. each variable before computing the gradient using this function. If `set_grad` option is set to `True`, the computed gradient is also stored in the `Variable.grad_var` attribute of each variable, in which case any original value of `Variable.grad_var` will be updated even if it had already been set.

Parameters

- **outputs** (tuple or list of *Variable*) – A sequence of output variables from which backprop starts.
- **inputs** (tuple or list of *Variable*) – A sequence of input variables each of which this function computes the gradient w.r.t.
- **grad_outputs** (tuple or list of *Variable* or None) – A sequence of variables that gives the initial value of each output gradient. If an element is set to `None`, an array filled with 1 is used. If this argument itself is `None`, it is treated as a sequence of `None`s.
- **grad_inputs** (tuple or list of *Variable* or None) – A sequence of variables that gives the initial value of each input gradient. The gradients computed by the backprop algorithm are accumulated to them (not in-place). If an element is set to `None`, the gradient is not accumulated to this value. If this argument itself is `None`, it is treated as a sequence of `None`s.
- **set_grad** (*bool*) – If it is `True`, the *Variable.grad_var* attribute of each input variable is set to the corresponding computed gradient variable.
- **retain_grad** (*bool*) – If it is `True`, the gradients w.r.t. all the intermediate variables are stored in the *Variable.grad_var* attribute. In this case, the `set_grad` option is ignored.
- **enable_double_backprop** (*bool*) – If it is `True`, the computed gradients can be further backpropagated. Enabling it may increase the memory consumption (and possibly the computational time) to remember the intermediate gradient values for the second backpropagation.
- **loss_scale** (*float*) – Loss scaling factor. Loss scaling is a usefull technique to mitigate vanishing gradient issue that tends to happen when low precision data type like float16 is used during training. If you set loss scaling factor, gradients of loss values are to be multiplied by the factor before backprop starts. The factor is propagated to whole gradients in a computational graph along the backprop. The gradients of parameters are divided by the factor just before the parameters are to be updated.

Returns A list of gradient variables w.r.t. the inputs.

4.2.13 Function hooks

Chainer provides a function-hook mechanism that enriches the behavior of forward and backward propagation of *FunctionNode* and *Function*.

chainer.function_hooks.CUDAProfileHook	
chainer.function_hooks.CupyMemoryProfileHook	Function hook for measuring memory usage of functions in cupy memory pool.
chainer.function_hooks.PrintHook	Function hook that prints debug information.
chainer.function_hooks.TimerHook	Function hook for measuring elapsed time of functions.

chainer.function_hooks.CUDAProfileHook

class `chainer.function_hooks.`**`CUDAProfileHook`**

Methods

__enter__()

__exit__(*_)

added(*function*)
Callback function invoked when the function hook is registered

Parameters **function** (FunctionNode) – Function object to which the function hook is added. None if the function hook is registered globally.

backward_postprocess(*function, in_data, out_grad*)
Callback function invoked after backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

backward_preprocess(*function, in_data, out_grad*)
Callback function invoked before backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

deleted(*function*)
Callback function invoked when the function hook is unregistered

Parameters **function** (FunctionNode) – Function object from which the function hook is deleted. None if the function hook was registered globally.

forward_postprocess(*function, in_data*)
Callback function invoked after forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

forward_preprocess(*function, in_data*)
Callback function invoked before forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'CUDAProfileHook'

chainer.function_hooks.CupyMemoryProfileHook

class chainer.function_hooks.**CupyMemoryProfileHook**
Function hook for measuring memory usage of functions in cupy memory pool.

Example

Code example:

```
from chainer.function_hooks import CupyMemoryProfileHook
hook = CupyMemoryProfileHook()
with hook:
    trainer.run()
hook.print_report()
```

Output example:

```
        FunctionName  UsedBytes  AcquiredBytes  Occurrence
      LinearFunction     5.16GB       179.98MB        3900
                ReLU     0.99GB       458.97MB        2600
SoftmaxCrossEntropy      0.01GB         5.08MB        1300
            Accuracy     0.00GB         0.35MB         700
```

where *FunctionName* is the name of function that calls the hook, and *UsedBytes* is the memory bytes the function used from cupy memory pool, and *AcquiredBytes* is the actual memory bytes the cupy memory pool acquired from GPU device on the function call, and *Occurrence* is the number of calls.

Variables **call_history** – List of measurement results. It consists of the name of the function that calls this hook, the memory bytes the function used from cupy memory pool, and the memory bytes the cupy memory pool acquired from GPU device on the function call.

Methods

__enter__()

__exit__(*_)

added (*function=None*)
Callback function invoked when the function hook is registered

Parameters **function** (FunctionNode) – Function object to which the function hook is added. None if the function hook is registered globally.

backward_postprocess (*function*, *in_data*, *out_grad*)
Callback function invoked after backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

backward_preprocess (*function*, *in_data*, *out_grad*)
Callback function invoked before backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

deleted (*function=None*)
Callback function invoked when the function hook is unregistered

Parameters **function** (FunctionNode) – Function object from which the function hook is deleted. None if the function hook was registered globally.

forward_postprocess (*function*, *in_data*)
Callback function invoked after forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

forward_preprocess (*function*, *in_data*)
Callback function invoked before forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

print_report (*unit='auto'*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Prints a summary report of memory profiling in functions.

Parameters **unit** (*str*) – Supplementary units used for used memories. *B*, *KB*, *MB*, *GB*, *TB*, *PB*, *EB*, *ZB*, *auto'(default) and 'auto_foreach* are supported. If *auto*, units of memories are aligned to the largest values of 'used_bytes' and 'acquired_bytes'. If *auto_foreach*, units of memories are adjusted for each element.

summary()
Returns a summary of memory profiling in functions.

Returns A summarized dictionary whose keys are function names and values are dictionaries of `used_bytes`, `acquired_bytes`, and `occurrrence`.

total_acquired_bytes()
Returns total bytes that cupy memory pool acquired from GPU.

total_used_bytes()
Returns total bytes that functions used from cupy memory pool.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'CupyMemoryProfileHook'

chainer.function_hooks.PrintHook

class chainer.function_hooks.**PrintHook**(*sep=None, end='n', file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, flush=True*)

Function hook that prints debug information.

This function hook outputs the debug information of input arguments of `forward` and `backward` methods involved in the hooked functions at preprocessing time (that is, just before each method is called).

Unlike simple "debug print" technique, where users insert print functions at every function to be inspected, we can show the information of all functions involved with single `with` statement.

Further, this hook enables us to show the information of `backward` methods without inserting print functions into Chainer's library code.

Parameters

- **sep** – *(deprecated since v4.0.0)* Ignored.
- **end** – Character to be added at the end of print function.
- **file** – Output file_like object that that redirect to.
- **flush** – If `True`, this hook forcibly flushes the text stream at the end of preprocessing.

Example

The basic usage is to use it with `with` statement.

```
>>> from chainer import function_hooks
>>> l = L.Linear(10, 10)
>>> x = chainer.Variable(np.zeros((1, 10), np.float32))
>>> with chainer.function_hooks.PrintHook():
...     y = l(x)
...     z = F.sum(y)
...     z.backward()
```

In this example, `PrintHook` shows the debug information of forward propagation of `LinearFunction` (which is implicitly called by `l`) and `Sum` (called by `F.sum`) and backward propagation of `z` and `y`.

Methods

__enter__()

__exit__(**_*)

added(*function*)
Callback function invoked when the function hook is registered

Parameters **function** (`FunctionNode`) – Function object to which the function hook is added. `None` if the function hook is registered globally.

backward_postprocess(*function*, *in_data*, *out_grad*)
Callback function invoked after backward propagation.

Parameters

- **function** (`FunctionNode`) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

backward_preprocess(*function*, *in_data*, *out_grad*)
Callback function invoked before backward propagation.

Parameters

- **function** (`FunctionNode`) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

deleted(*function*)
Callback function invoked when the function hook is unregistered

Parameters **function** (`FunctionNode`) – Function object from which the function hook is deleted. `None` if the function hook was registered globally.

forward_postprocess(*function*, *in_data*)
Callback function invoked after forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

forward_preprocess (*function*, *in_data*)
Callback function invoked before forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'PrintHook'

chainer.function_hooks.TimerHook

class chainer.function_hooks.**TimerHook**
Function hook for measuring elapsed time of functions.

Example

Code example:

```
from chainer.function_hooks import TimerHook
hook = TimerHook()
with hook:
    trainer.run()
hook.print_report()
```

Output example:

```
       FunctionName  ElapsedTime  Occurrence
     LinearFunction      1.24sec        3900
               ReLU      0.59sec        2600
```

(continues on next page)

(continued from previous page)

```
SoftmaxCrossEntropy       0.82sec        1300
           Accuracy       0.18sec         700
```

where *FunctionName* is the name of function that calls the hook, and *ElapsedTime* is the elapsed time the function consumed, and *Occurrence* is the number of calls.

Variables **call_history** – List of measurement results. It consists of pairs of the name of the function that calls this hook and the elapsed time the function consumes.

Methods

__enter__()

__exit__(*_)

added(*function*)

Callback function invoked when the function hook is registered

Parameters **function** (FunctionNode) – Function object to which the function hook is added. None if the function hook is registered globally.

backward_postprocess(*function*, *in_data*, *out_grad*)

Callback function invoked after backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

backward_preprocess(*function*, *in_data*, *out_grad*)

Callback function invoked before backward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

deleted(*function*)

Callback function invoked when the function hook is unregistered

Parameters **function** (FunctionNode) – Function object from which the function hook is deleted. None if the function hook was registered globally.

forward_postprocess(*function*, *in_data*)

Callback function invoked after forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

forward_preprocess(*function*, *in_data*)

Callback function invoked before forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

print_report(*unit='auto'*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Prints a summary report of time profiling in functions.

Parameters **unit** (*str*) – Supplementary units used for computational times. *sec*, *ms*, *us*, *ns*, *auto'(default) and 'auto_foreach* are supported. If *auto*, units of times are aligned to the largest, and if *auto_foreach*, units of times are adjusted for each element.

summary()
Returns a summary of time profiling in functions.

Returns A summarized dictionary whose keys are function names and values are dictionaries of *elapsed_time* and *occurrence*.

total_time()
Returns total elapsed time in seconds.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'TimerHook'

table = {'ms': 1000, 'ns': 1000000000, 'sec': 1, 'us': 1000000}

You can also implement your own function-hook to inject arbitrary code before/after the forward/backward propagation.

chainer.FunctionHook	Base class of hooks for Functions.

chainer.FunctionHook

class chainer.**FunctionHook**
Base class of hooks for Functions.

FunctionHook is a callback object that is registered to *FunctionNode*. Registered function hooks are invoked before and after forward and backward operations of each function.

Function hooks that derive from *FunctionHook* may override the following methods:

- *added()*
- *deleted()*
- *forward_preprocess()*
- *forward_postprocess()*
- *backward_preprocess()*
- *backward_postprocess()*

By default, these methods do nothing.

Specifically, when the *__call__()* method of some function is invoked, *forward_preprocess()* (resp. *forward_postprocess()*) of all function hooks registered to this function are called before (resp. after) forward propagation.

Likewise, when *backward()* of some *Variable* is invoked, *backward_preprocess()* (resp. *backward_postprocess()*) of all function hooks registered to the function which holds this variable as a gradient are called before (resp. after) backward propagation.

added() and *deleted()* are called when the hook is registered or unregistered, respectively.

There are two ways to register *FunctionHook* objects to *FunctionNode* objects.

The first one is to use `with` statement. Function hooks hooked in this way are registered to all functions within `with` statement and are unregistered at the end of `with` statement.

Example

The following code is a simple example in which we measure the elapsed time of a part of forward propagation procedure with *TimerHook*, which is a subclass of *FunctionHook*.

```
>>> class Model(chainer.Chain):
...   def __init__(self):
...     super(Model, self).__init__()
...     with self.init_scope():
...       self.l = L.Linear(10, 10)
...   def __call__(self, x1):
...     return F.exp(self.l(x1))
>>> model1 = Model()
>>> model2 = Model()
>>> x = chainer.Variable(np.zeros((1, 10), np.float32))
>>> with chainer.function_hooks.TimerHook() as m:
...   _ = model1(x)
...   y = model2(x)
>>> model3 = Model()
>>> z = model3(y)
>>> print('Total time : {}'.format(m.total_time()))
...
Total time : ...
```

In this example, we measure the elapsed times for each forward propagation of all functions in `model1` and `model2`. Note that `model3` is not a target of measurement as *TimerHook* is unregistered before forward propagation of `model3`.

Note: Chainer stores the dictionary of registered function hooks as a thread local object. So, function hooks registered are different depending on threads.

The other one is to register it directly to a `FunctionNode` object by calling its `add_hook()` method. Function hooks registered in this way can be removed by `delete_hook()` method. Contrary to the former registration method, function hooks are registered only to the function whose `add_hook()` method is called.

If the hook is registered globally using `with` statement, `None` is passed as the `function` argument of `added()` and `deleted()`.

If the hook is registered in a specific function using `add_hook()`, the `FunctionNode` instance is passed as the `function` argument of `added()` and `deleted()`.

Parameters **name** (`str`) – Name of this function hook.

Methods

`__enter__`()

`__exit__`(*_)

`added`(*function*)

Callback function invoked when the function hook is registered

Parameters **function** (`FunctionNode`) – Function object to which the function hook is added. `None` if the function hook is registered globally.

`backward_postprocess`(*function, in_data, out_grad*)

Callback function invoked after backward propagation.

Parameters

- **function** (`FunctionNode`) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

`backward_preprocess`(*function, in_data, out_grad*)

Callback function invoked before backward propagation.

Parameters

- **function** (`FunctionNode`) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.
- **out_grad** (tuple of *N-dimensional array*) – Gradient data of backward propagation.

`deleted`(*function*)

Callback function invoked when the function hook is unregistered

Parameters **function** (`FunctionNode`) – Function object from which the function hook is deleted. `None` if the function hook was registered globally.

`forward_postprocess`(*function, in_data*)

Callback function invoked after forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

forward_preprocess (*function*, *in_data*)
Callback function invoked before forward propagation.

Parameters

- **function** (FunctionNode) – Function object to which the function hook is registered.
- **in_data** (tuple of *N-dimensional array*) – Input data of forward propagation.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'FunctionHook'

4.3 Link and Chains

Chainer provides many *Link* implementations in the *chainer.links* package.

Note: Some of the links are originally defined in the *chainer.functions* namespace. They are still left in the namespace for backward compatibility, though it is strongly recommended that you use them via the *chainer.links* package.

4.3.1 Learnable connections

chainer.links.Bias	Broadcasted elementwise summation with learnable parameters.
chainer.links.Bilinear	Bilinear layer that performs tensor multiplication.
chainer.links.ChildSumTreeLSTM	Child-Sum TreeLSTM unit.
chainer.links.Convolution1D	1-dimensional convolution layer.
chainer.links.Convolution2D	Two-dimensional convolutional layer.

Continued on next page

Table 16 – continued from previous page

`chainer.links.Convolution3D`	3-dimensional convolution layer.
`chainer.links.ConvolutionND`	N-dimensional convolution layer.
`chainer.links.Deconvolution1D`	1-dimensional deconvolution layer.
`chainer.links.Deconvolution2D`	Two dimensional deconvolution function.
`chainer.links.Deconvolution3D`	3-dimensional deconvolution layer.
`chainer.links.DeconvolutionND`	N-dimensional deconvolution function.
`chainer.links.DeformableConvolution2D`	Two-dimensional deformable convolutional layer.
`chainer.links.DepthwiseConvolution2D`	Two-dimensional depthwise convolutional layer.
`chainer.links.DilatedConvolution2D`	Two-dimensional dilated convolutional layer.
`chainer.links.EmbedID`	Efficient linear layer for one-hot input.
`chainer.links.GRU`	Stateful Gated Recurrent Unit function (GRU)
`chainer.links.Highway`	Highway module.
`chainer.links.Inception`	Inception module of GoogLeNet.
`chainer.links.InceptionBN`	Inception module of the new GoogLeNet with BatchNormalization.
`chainer.links.Linear`	Linear layer (a.k.a. fully-connected layer).
`chainer.links.LocalConvolution2D`	Two-dimensional local convolutional layer.
`chainer.links.LSTM`	Fully-connected LSTM layer.
`chainer.links.MLPConvolution2D`	Two-dimensional MLP convolution layer of Network in Network.
`chainer.links.NaryTreeLSTM`	N-ary TreeLSTM unit.
`chainer.links.NStepBiGRU`	Stacked Bi-directional GRU for sequences.
`chainer.links.NStepBiLSTM`	Stacked Bi-directional LSTM for sequences.
`chainer.links.NStepBiRNNReLU`	Stacked Bi-directional RNN for sequences.
`chainer.links.NStepBiRNNTanh`	Stacked Bi-directional RNN for sequences.
`chainer.links.NStepGRU`	Stacked Uni-directional GRU for sequences.
`chainer.links.NStepLSTM`	Stacked Uni-directional LSTM for sequences.
`chainer.links.NStepRNNReLU`	Stacked Uni-directional RNN for sequences.
`chainer.links.NStepRNNTanh`	Stacked Uni-directional RNN for sequences.
`chainer.links.Parameter`	Link that just holds a parameter and returns it.
`chainer.links.Scale`	Broadcasted elementwise product with learnable parameters.
`chainer.links.StatefulGRU`	Stateful Gated Recurrent Unit function (GRU).
`chainer.links.StatelessGRU`	Stateless Gated Recurrent Unit function (GRU).
`chainer.links.StatefulMGU`	
`chainer.links.StatelessMGU`	
`chainer.links.StatefulPeepholeLSTM`	Fully-connected LSTM layer with peephole connections.
`chainer.links.StatefulZoneoutLSTM`	
`chainer.links.StatelessLSTM`	Stateless LSTM layer.

chainer.links.Bias

class `chainer.links.`**`Bias`**(*axis=1*, *shape=None*)

Broadcasted elementwise summation with learnable parameters.

Computes a elementwise summation as `bias()` function does except that its second input is a learnable bias parameter b the link has.

Parameters

- **axis** (*int*) – The first axis of the first input of `bias()` function along which its second input is applied.

- **shape** (*tuple of ints*) – Shape of the learnable bias parameter. If `None`, this link does not have learnable parameters so an explicit bias needs to be given to its `forward` method's second input.

See also:

See *bias()* for details.

Variables b (`Variable`) – Bias parameter if `shape` is given. Otherwise, no attributes.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters link (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**xs*)
Applies broadcasted elementwise summation.

Parameters xs (*list of Variables*) – Input variables whose length should be one if the link has a learnable bias parameter, otherwise should be two.

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all

elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *`Sequential`* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *`get_device()`* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Bilinear

class `chainer.links.`**Bilinear**(*left_size*, *right_size*, *out_size*, *nobias=False*, *initialW=None*, *initial_bias=None*)
Bilinear layer that performs tensor multiplication.

Bilinear is a primitive link that wraps the `bilinear()` functions. It holds parameters `W`, `V1`, `V2`, and `b` corresponding to the arguments of `bilinear()`.

Parameters

- **left_size** (*int*) – Dimension of input vector e^1 (J)
- **right_size** (*int*) – Dimension of input vector e^2 (K)
- **out_size** (*int*) – Dimension of output vector y (L)
- **nobias** (*bool*) – If True, parameters V1, V2, and b are omitted.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is numpy.ndarray, its ndim should be 3.
- **initial_bias** (tuple of *initializer*) – Initial values of V^1, V^2 and b. The length of this argument must be 3. Each element of this tuple must have the shapes of (left_size, out_size), (right_size, out_size), and (out_size,), respectively if they are numpy.ndarray. If None, V^1 and V^2 are initialized by the default initializer and b is set to 0.

See also:

See chainer.functions.bilinear() for details.

Variables

- **W** (Variable) – Bilinear weight parameter.
- **V1** (Variable) – Linear weight parameter for the first argument.
- **V2** (Variable) – Linear weight parameter for the second argument.
- **b** (Variable) – Bias parameter.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a

scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise,

it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*e1*, *e2*)

Applies the bilinear function to inputs and the internal parameters.

Parameters

- **e1** (`Variable`) – Left input.
- **e2** (`Variable`) – Right input.

Returns Output variable.

Return type *Variable*

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zero_grads()

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

`Device` instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.ChildSumTreeLSTM

class `chainer.links.ChildSumTreeLSTM`(*in_size*, *out_size*)
Child-Sum TreeLSTM unit.

> **Warning:** This feature is experimental. The interface can change in the future.

This is a Child-Sum TreeLSTM unit as a chain. This link is a variable arguments function, which compounds the states of all children nodes into the new states of a current (parent) node. *states* denotes the cell state, c, and the output, h, which are produced by this link. This link doesn't keep cell and hidden states internally.

For example, this link is called such as `func(c1, c2, h1, h2, x)` if the number of children nodes is 2, while `func(c1, c2, c3, h1, h2, h3, x)` if that is 3. This function is *independent* from an order of children nodes. Thus, the returns of `func(c1, c2, h1, h2, x)` equal to those of `func(c2, c1, h2, h1, x)`.

Parameters

- **in_size** (*int*) – Dimension of input vectors.
- **out_size** (*int*) – Dimensionality of cell and output vectors.

Variables

- **W_x** (`chainer.links.Linear`) – Linear layer of connections from input vectors.
- **W_h_aio** (`chainer.links.Linear`) – Linear layer of connections between (a, i, o) and summation of children's output vectors. a, i and o denotes input compound, input gate and output gate, respectively. a, input compound, equals to u in the paper by Tai et al.
- **W_h_f** (`chainer.links.Linear`) – Linear layer of connections between forget gate f and the output of each child.

See the paper for details: Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*`str`*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*`str`*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*`int or tuple of ints`*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*`str`*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

`addgrads`(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**cshsx*)
Returns new cell state and output of Child-Sum TreeLSTM.

Parameters **cshsx** (list of *Variable*) – Variable arguments which include all cell vectors and all output vectors of variable children, and an input vector.

Returns Returns (c_{new}, h_{new}), where c_{new} represents new cell state vector, and h_{new} is new output vector.

Return type tuple of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)

Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)

Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Convolution1D

class chainer.links.**Convolution1D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *nobias=False*, *initialW=None*, *initial_bias=None*, *cover_all=False*, *dilate=1*, *groups=1*)

1-dimensional convolution layer.

Note: This link wraps *ConvolutionND* by giving 1 to the first argument `ndim`, so see the details of the behavior in the documentation of *ConvolutionND*.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies N-dimensional convolution layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of convolution.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override device_resident_accept() to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override device_resident_accept() to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use to_device to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Convolution2D

class `chainer.links.`**`Convolution2D`**(*self*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *nobias=False*, *initialW=None*, *initial_bias=None*, ***, *dilate=1*, *groups=1*)

Two-dimensional convolutional layer.

This link wraps the `convolution_2d()` function and holds the filter weight and bias vector as parameters.

The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.deterministic` is `True` and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

Convolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **`in_channels`** (*`int` or `None`*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **`out_channels`** (*`int`*) – Number of channels of output arrays.
- **`ksize`** (*`int` or pair of `int`s*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **`stride`** (*`int` or pair of `int`s*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **`pad`** (*`int` or pair of `int`s*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **`nobias`** (*`bool`*) – If `True`, then this link does not use the bias term.
- **`initialW`** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 4.
- **`initial_bias`** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.
- **`dilate`** (*`int` or pair of `int`s*) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **`groups`** (`int`) – Number of groups of channels. If the number is greater than 1, input tensor W is divided into some blocks by this value channel-wise. For each tensor blocks, convolution operation will be executed independently. Input channel size `in_channels` and output channel size `out_channels` must be exactly divisible by this value.

See also:

See `chainer.functions.convolution_2d()` for the definition of two-dimensional convolution.

Variables

- **`W`** (`Variable`) – Weight parameter.
- **`b`** (`Variable`) – Bias parameter.

Example

There are several ways to make a Convolution2D link.

Let an input vector `x` be:

```
>>> x = np.arange(1 * 3 * 10 * 10, dtype=np.float32).reshape(
...     1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

```
>>> l = L.Convolution2D(3, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

2. Omit `in_channels` or fill it with `None`:

   The below two cases are the same.

```
>>> l = L.Convolution2D(7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

```
>>> l = L.Convolution2D(None, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

   When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword auguments. So the below two cases are the same.

```
>>> l = L.Convolution2D(7, 5, stride=1, pad=0)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

```
>>> l = L.Convolution2D(None, 7, 5, 1, 0)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (`str`) – Name of the parameter. This name is also used as the attribute name.
- **shape** (`int or tuple of ints`) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (`str`) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters link (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters mode (`str`) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies the convolution layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of the convolution.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)
Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

`to_device`(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **`device`** – Target device specifier. See *`get_device()`* for available values.

Returns: self

`to_gpu`(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **`device`** – Target device specifier. If omitted, the current device is used.

Returns: self

`to_intel64`()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

`zerograds`()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *`cleargrads()`* instead.

`__eq__`()

Return self==value.

`__ne__`()

Return self!=value.

`__lt__`()

Return self<value.

`__le__`()

Return self<=value.

`__gt__`()

Return self>value.

`__ge__`()

Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Convolution3D

class `chainer.links.Convolution3D`(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *nobias=False*, *initialW=None*, *initial_bias=None*, *cover_all=False*, *dilate=1*, *groups=1*)

3-dimensional convolution layer.

Note: This link wraps `ConvolutionND` by giving 3 to the first argument `ndim`, so see the details of the behavior in the documentation of `ConvolutionND`.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (`str`) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

Applies N-dimensional convolution layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of convolution.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (`str`) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.ConvolutionND

class `chainer.links.`**ConvolutionND**(*ndim*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *nobias=False*, *initialW=None*, *initial_bias=None*, *cover_all=False*, *dilate=1*, *groups=1*)

N-dimensional convolution layer.

This link wraps the `convolution_nd()` function and holds the filter weight and bias vector as parameters.

Convolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **ndim** (*int*) – Number of spatial dimensions.
- **in_channels** (*int*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or tuple of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k, ..., k)` are equivalent.
- **stride** (*int or tuple of ints*) – Stride of filter application. `stride=s` and `stride=(s, s, ..., s)` are equivalent.
- **pad** (*int or tuple of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p, ..., p)` are equivalent.

- **nobias** (*bool*) – If True, then this function does not use the bias.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is numpy.ndarray, its ndim should be $n + 2$ where n is the number of spatial dimensions.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If None, the bias will be initialized to zero. When it is numpy.ndarray, its ndim should 1.
- **cover_all** (*bool*) – If True, all spatial locations are convoluted into some output pixels. It may make the output size larger. cover_all needs to be False if you want to use cuDNN.
- **dilate** (int or tuple of int s) – Dilation factor of filter applications. dilate=d and dilate=(d, d, ..., d) are equivalent.
- **groups** (int) – The number of groups to use grouped convolution. The default is one, where grouped convolution is not used.

See also:

See *convolution_nd()* for the definition of N-dimensional convolution. See *convolution_2d()* for the definition of two-dimensional convolution.

Variables

- **W** (Variable) – Weight parameter.
- **b** (Variable) – Bias parameter. If initial_bias is None, set to None.

Example

There are several ways to make a ConvolutionND link.

Let an input vector x be:

```
>>> x = np.arange(2 * 5 * 5 * 5, dtype=np.float32).reshape(
...     1, 2, 5, 5, 5)
```

1. Give the first four arguments explicitly:

```
>>> l = L.ConvolutionND(3, 2, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2, 2)
```

2. Omit in_channels or fill it with None:

   The below two cases are the same.

```
>>> l = L.ConvolutionND(3, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2, 2)
```

```
>>> l = L.ConvolutionND(3, None, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2, 2)
```

When you omit the second argument, you need to specify the other subsequent arguments from `stride` as keyword auguments. So the below two cases are the same.

```
>>> l = L.ConvolutionND(3, 7, 4, stride=1, pad=0)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2, 2)
```

```
>>> l = L.ConvolutionND(3, None, 7, 4, 1, 0)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2, 2)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*`str`*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *`initialize()`* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link, copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *`BatchNormalization`*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*`bool`*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *`Parameter`*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*x*)
Applies N-dimensional convolution layer.

Parameters **x** (Variable) – Input image.

Returns Output of convolution.

Return type *Variable*

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to

others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Deconvolution1D

class chainer.links.**Deconvolution1D**(*in_channels, out_channels, ksize, stride=1, pad=0, nobias=False, outsize=None, initialW=None, initial_bias=None, dilate=1, groups=1*)

1-dimensional deconvolution layer.

Note: This link wraps *DeconvolutionND* by giving 1 to the first argument ndim, so see the details of the behavior in the documentation of *DeconvolutionND*.

Methods

__call__(**args, **kwargs*)
Call self as a function.

add_hook(*hook, name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_param(*name, shape=None, dtype=<class 'numpy.float32'>, initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name, value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*x*)

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all

elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Attributes

`device`
`Device` instance.

`local_link_hooks`
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

`printable_specs`
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

`update_enabled`
`True` if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

`xp`
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Deconvolution2D

class `chainer.links.`**`Deconvolution2D`**(*self*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *nobias=False*, *outsize=None*, *initialW=None*, *initial_bias=None*, ***, *dilate=1*, *groups=1*)
Two dimensional deconvolution function.

This link wraps the `deconvolution_2d()` function and holds the filter weight and bias vector as parameters.

Deconvolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **in_channels** (*int or None*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **nobias** (*bool*) – If `True`, then this function does not use the bias term.
- **outsize** (*tuple*) – Expected output size of deconvolutional operation. It should be pair of height and width (out_H, out_W). Default value is `None` and the outsize is estimated by input size, stride and pad.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 4.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.
- **dilate** (`int` or `tuple` of `int` s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **groups** (*int*) – The number of groups to use grouped deconvolution. The default is one, where grouped deconvolution is not used.

The filter weight has four dimensions (c_I, c_O, k_H, k_W) which indicate the number of input channels, output channels, height and width of the kernels, respectively. The filter weight is initialized with i.i.d. Gaussian random samples, each of which has zero mean and deviation $\sqrt{1/(c_I k_H k_W)}$ by default.

The bias vector is of size c_O. Its elements are initialized by `bias` argument. If `nobias` argument is set to True, then this function does not hold the bias parameter.

The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.cudnn_deterministic` is `True` and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

See also:

See `chainer.functions.deconvolution_2d()` for the definition of two-dimensional convolution.

See also:

See `chainer.links.Convolution2D()` for the examples of ways to give arguments to this link.

Example

There are several ways to make a Deconvolution2D link.

Let an input vector `x` be:

```
>>> x = np.arange(1 * 3 * 10 * 10, dtype=np.float32).reshape(
...     1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

   In this case, all the other arguments are set to the default values.

   ```
   >>> l = L.Deconvolution2D(3, 7, 4)
   >>> y = l(x)
   >>> y.shape
   (1, 7, 13, 13)
   ```

2. Omit `in_channels` or fill it with `None`:

   The below two cases are the same.

   ```
   >>> l = L.Deconvolution2D(7, 4)
   >>> y = l(x)
   >>> y.shape
   (1, 7, 13, 13)
   ```

   ```
   >>> l = L.Deconvolution2D(None, 7, 4)
   >>> y = l(x)
   >>> y.shape
   (1, 7, 13, 13)
   ```

   When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword arguments. So the below two cases are the same.

   ```
   >>> l = L.Deconvolution2D(None, 7, 4, 2, 1)
   >>> y = l(x)
   >>> y.shape
   (1, 7, 20, 20)
   ```

   ```
   >>> l = L.Deconvolution2D(7, 4, stride=2, pad=1)
   >>> y = l(x)
   >>> y.shape
   (1, 7, 20, 20)
   ```

Methods

`__call__`(**args, **kwargs*)
Call self as a function.

`add_hook`(*hook, name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (*Link*) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Deconvolution3D

class `chainer.links.`**Deconvolution3D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *nobias=False*, *outsize=None*, *initialW=None*, *initial_bias=None*, *dilate=1*, *groups=1*)
3-dimensional deconvolution layer.

Note: This link wraps `DeconvolutionND` by giving 3 to the first argument `ndim`, so see the details of the behavior in the documentation of `DeconvolutionND`.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (`str`) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (`str`) – Name of the parameter. This name is also used as the attribute name.

- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default mode is share.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters name (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *`Parameter`* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

`links`(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **`skipself`** (*`bool`*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

`namedlinks`(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **`skipself`** (*`bool`*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

`namedparams`(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **`include_uninit`** (*`bool`*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

`params`(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **`include_uninit`** (*`bool`*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

`register_persistent`(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **`name`** (*`str`*) – Name of the attribute to be registered.

`repeat`(*n_repeat, mode='init'*)
Repeats this link multiple times to make a *`Sequential`*.

This method returns a *`Sequential`* object which has the same *`Link`* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *`Sequential`* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.DeconvolutionND

class chainer.links.**DeconvolutionND**(*ndim*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *nobias=False*, *outsize=None*, *initialW=None*, *initial_bias=None*, *dilate=1*, *groups=1*)

N-dimensional deconvolution function.

This link wraps `deconvolution_nd()` function and holds the filter weight and bias vector as its parameters.

Deconvolution links can use a feature of cuDNN called autotuning, which selects the most efficient CNN algorithm for images of fixed-size, can provide a significant performance boost for fixed neural nets. To enable, set *chainer.using_config('autotune', True)*

Parameters

- **ndim** (*int*) – Number of spatial dimensions.
- **in_channels** (*int*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or tuple of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k, ..., k)` are equivalent.
- **stride** (*int or tuple of ints*) – Stride of filter application. `stride=s` and `stride=(s, s, ..., s)` are equivalent.
- **pad** (*int or tuple of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p, ..., p)` are equivalent.
- **nobias** (*bool*) – If `True`, then this function does not use the bias.
- **outsize** (*tuple of ints*) – Expected output size of deconvolutional operation. It should be a tuple of ints that represents the output size of each dimension. Default value is `None` and the outsize is estimated with input size, stride and pad.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be $n + 2$ where n is the number of spatial dimensions.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should 1.

- **dilate** (int or tuple of int s) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d, ..., d)` are equivalent.
- **groups** (int) – The number of groups to use grouped convolution. The default is one, where grouped convolution is not used.

See also:

deconvolution_nd()

Variables

- **W** (Variable) – Weight parameter.
- **b** (Variable) – Bias parameter. If `initial_bias` is `None`, set to `None`.

Example

There are several ways to make a DeconvolutionND link.

Let an input vector `x` be:

```
>>> x = np.arange(2 * 5 * 5 * 5, dtype=np.float32).reshape(
...     1, 2, 5, 5, 5)
```

1. Give the first four arguments explicitly:

```
>>> l = L.DeconvolutionND(3, 2, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 8, 8, 8)
```

2. Omit `in_channels` or fill it with `None`:

   The below two cases are the same.

```
>>> l = L.DeconvolutionND(3, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 8, 8, 8)
```

```
>>> l = L.DeconvolutionND(3, None, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 8, 8, 8)
```

   When you omit the second argument, you need to specify the other subsequent arguments from `stride` as keyword auguments. So the below two cases are the same.

```
>>> l = L.DeconvolutionND(3, 7, 4, stride=2, pad=1)
>>> y = l(x)
>>> y.shape
(1, 7, 10, 10, 10)
```

```
>>> l = L.DeconvolutionND(3, None, 7, 4, 2, 1)
>>> y = l(x)
>>> y.shape
(1, 7, 10, 10, 10)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*`str`*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*`int or tuple of ints`*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*`str`*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

`addgrads`(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

`children`()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.DeformableConvolution2D

class chainer.links.**DeformableConvolution2D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *offset_nobias=False*, *offset_initialW=None*, *offset_initial_bias=None*, *deform_nobias=False*, *deform_initialW=None*, *deform_initial_bias=None*)

Two-dimensional deformable convolutional layer.

This link wraps the convolution layer for offset prediction and the *deformable_convolution_2d_sampler()* function. This also holds the filter weights and bias vectors of two convolution layers as parameters.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **offset_nobias** (*bool*) – If True, then this link does not use the bias term for the first convolution layer.
- **offset_initialW** (*initializer*) – Initializer to initialize the weight of the first convolution layer. When it is `numpy.ndarray`, its `ndim` should be 4.
- **offset_initial_bias** (*initializer*) – Initializer to initialize the bias of the first convolution layer. If None, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.
- **deform_nobias** (*bool*) – If True, then this link does not use the bias term for the second convolution layer.
- **deform_initialW** (*initializer*) – Initializer to initialize the weight for the second convolution layer. When it is `numpy.ndarray`, its `ndim` should be 4.
- **deform_initial_bias** (*initializer*) – Initializer to initialize the bias for the second convolution layer. If None, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.

See also:

See `chainer.functions.deformable_convolution_2d_sampler()`.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies the deformable convolution.

Parameters **x** (`Variable`) – Input image.

Returns Output of the deformable convolution.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.DepthwiseConvolution2D

class `chainer.links.`**`DepthwiseConvolution2D`**(*in_channels*, *channel_multiplier*, *ksize*, *stride=1*, *pad=0*, *nobias=False*, *initialW=None*, *initial_bias=None*)

Two-dimensional depthwise convolutional layer.

This link wraps the `depthwise_convolution_2d()` function and holds the filter weight and bias vector as parameters.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **channel_multiplier** (*int*) – Channel multiplier number. Number of output arrays equal `in_channels * channel_multiplier`.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **nobias** (*bool*) – If `True`, then this link does not use the bias term.

- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 4.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.

See also:

See `chainer.functions.depthwise_convolution_2d()`.

Variables

- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*x*)
Applies the depthwise convolution layer.

Parameters **x** (chainer.Variable or `numpy.ndarray` or cupy.ndarray) – Input image.

Returns Output of the depthwise convolution.

Return type *Variable*

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to

others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.DilatedConvolution2D

class chainer.links.**DilatedConvolution2D**(*in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *dilate=1*, *nobias=False*, *initialW=None*, *initial_bias=None*)

Two-dimensional dilated convolutional layer.

This link wraps the *dilated_convolution_2d()* function and holds the filter weight and bias vector as parameters.

Note: You can also define a dilated convolutional layer by passing `dilate` argument to *chainer.links.Convolution2D*. The functionality is the same.

Parameters

- **in_channels** (*int or None*) – Number of channels of input arrays. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- **dilate** (*int or pair of ints*) – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.
- **nobias** (*bool*) – If `True`, then this link does not use the bias term.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 4.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.

See also:

See *chainer.functions.dilated_convolution_2d()* for the definition of two-dimensional dilated convolution.

Variables

- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

Example

There are several ways to make a DilatedConvolution2D link.

Let an input vector `x` be:

```
>>> x = np.arange(1 * 3 * 10 * 10, dtype=np.float32).reshape(1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

```
>>> l = L.DilatedConvolution2D(3, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

2. Omit `in_channels` or fill it with `None`:

   The below two cases are the same.

```
>>> l = L.DilatedConvolution2D(7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

```
>>> l = L.DilatedConvolution2D(None, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

   When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword auguments. So the below two cases are the same.

```
>>> l = L.DilatedConvolution2D(None, 7, 5, 1, 0, 2)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2)
```

```
>>> l = L.DilatedConvolution2D(7, 5, stride=1, pad=0, dilate=2)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters link (`Link`) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters mode (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies the convolution layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of the convolution.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)

Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.EmbedID

class `chainer.links.EmbedID`(*in_size*, *out_size*, *initialW=None*, *ignore_label=None*)

Efficient linear layer for one-hot input.

This is a link that wraps the `embed_id()` function. This link holds the ID (word) embedding matrix `W` as a parameter.

Parameters

- **in_size** (`int`) – Number of different identifiers (a.k.a. vocabulary size).
- **out_size** (`int`) – Size of embedding vector.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 2.
- **ignore_label** (`int or None`) – If `ignore_label` is an int value, `i`-th row of return value is filled with `0`.

See also:

`embed_id()`

Variables **W** (`Variable`) – Embedding parameter matrix.

Example

```
>>> W = np.array([[0, 0, 0],
...               [1, 1, 1],
...               [2, 2, 2]]).astype(np.float32)
>>> W
array([[0., 0., 0.],
       [1., 1., 1.],
```

(continues on next page)

(continued from previous page)

```
       [2., 2., 2.]], dtype=float32)
>>> l = L.EmbedID(W.shape[0], W.shape[1], initialW=W)
>>> x = np.array([2, 1]).astype(np.int32)
>>> x
array([2, 1], dtype=int32)
>>> y = l(x)
>>> y.array
array([[2., 2., 2.],
       [1., 1., 1.]], dtype=float32)
```

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Extracts the word embedding of given IDs.

Parameters **x** (Variable) – Batch vectors of IDs.

Returns Batch of corresponding embeddings.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

ignore_label = None

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.GRU

class chainer.links.**GRU**(*in_size*, *out_size*, *init=None*, *inner_init=None*, *bias_init=0*)

Stateful Gated Recurrent Unit function (GRU)

This is an alias of `StatefulGRU`.

Methods

`__call__`(**args*, ***kwargs*)

Call self as a function.

`__getitem__`(*name*)

Equivalent to getattr.

`add_hook`(*hook*, *name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (`str`) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)

Registers a child link to this chain.

Parameters

- **name** (`str`) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)

Registers a parameter to the link.

Parameters

- **name** (`str`) – Name of the parameter. This name is also used as the attribute name.
- **shape** (`int or tuple of ints`) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (`str`) – Name of the persistent value. This name is also used for the attribute name.

- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*self*, *x*)

Does forward propagation.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

set_state(*h*)

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Highway

class chainer.links.**Highway**(*in_out_size*, *nobias=False*, *activate=<function relu>*, *init_Wh=None*, *init_Wt=None*, *init_bh=None*, *init_bt=-1*)

Highway module.

In highway network, two gates are added to the ordinal non-linear transformation ($H(x) = activate(W_h x + b_h)$). One gate is the transform gate $T(x) = \sigma(W_t x + b_t)$, and the other is the carry gate $C(x)$. For simplicity, the author defined $C = 1 - T$. Highway module returns y defined as

$$y = activate(W_h x + b_h) \odot \sigma(W_t x + b_t) + x \odot (1 - \sigma(W_t x + b_t))$$

The output array has the same spatial size as the input. In order to satisfy this, W_h and W_t must be square matrices.

Parameters

- **in_out_size** (*int*) – Dimension of input and output vectors.
- **nobias** (*bool*) – If `True`, then this function does not use the bias.
- **activate** – Activation function of plain array. $tanh$ is also available.
- **init_Wh** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 2.
- **init_bh** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1.
- **init_Wt** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 2.
- **init_bt** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 1. Negative value is recommended by the author of the paper. (e.g. -1, -3, . . .).

See: Highway Networks.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

Computes the output of the Highway module.

Parameters **x** (`Variable`) – Input variable.

Returns Output variable. Its array has the same spatial size and the same minibatch size as the input array.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Inception

class chainer.links.**Inception**(*in_channels*, *out1*, *proj3*, *out3*, *proj5*, *out5*, *proj_pool*, *conv_init=None*, *bias_init=None*)

Inception module of GoogLeNet.

It applies four different functions to the input array and concatenates their outputs along the channel dimension. Three of them are 2D convolutions of sizes 1x1, 3x3 and 5x5. Convolution paths of 3x3 and 5x5 sizes have 1x1 convolutions (called projections) ahead of them. The other path consists of 1x1 convolution (projection) and 3x3 max pooling.

The output array has the same spatial size as the input. In order to satisfy this, Inception module uses appropriate padding for each convolution and pooling.

See: Going Deeper with Convolutions.

Parameters

- **in_channels** (*int or None*) – Number of channels of input arrays.
- **out1** (*int*) – Output size of 1x1 convolution path.
- **proj3** (*int*) – Projection size of 3x3 convolution path.
- **out3** (*int*) – Output size of 3x3 convolution path.
- **proj5** (*int*) – Projection size of 5x5 convolution path.
- **out5** (*int*) – Output size of 5x5 convolution path.
- **proj_pool** (*int*) – Projection size of max pooling path.

- **conv_init** (*initializer*) – Initializer to initialize the convolution matrix weights. When it is `numpy.ndarray`, its `ndim` should be 4.
- **bias_init** (*initializer*) – Initializer to initialize the convolution matrix weights. When it is `numpy.ndarray`, its `ndim` should be 1.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Computes the output of the Inception module.

Parameters **x** (Variable) – Input variable.

Returns Output variable. Its array has the same spatial size and the same minibatch size as the input array. The channel dimension has size `out1 + out3 + out5 + proj_pool`.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same

values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.InceptionBN

class chainer.links.**InceptionBN**(*in_channels*, *out1*, *proj3*, *out3*, *proj33*, *out33*, *pooltype*, *proj_pool=None*, *stride=1*, *conv_init=None*, *dtype=None*)

Inception module of the new GoogLeNet with BatchNormalization.

This chain acts like *Inception*, while InceptionBN uses the *BatchNormalization* on top of each convolution, the 5x5 convolution path is replaced by two consecutive 3x3 convolution applications, and the pooling method is configurable.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

Parameters

- **in_channels** (*int or None*) – Number of channels of input arrays.
- **out1** (*int*) – Output size of the 1x1 convolution path.
- **proj3** (*int*) – Projection size of the single 3x3 convolution path.
- **out3** (*int*) – Output size of the single 3x3 convolution path.
- **proj33** (*int*) – Projection size of the double 3x3 convolutions path.
- **out33** (*int*) – Output size of the double 3x3 convolutions path.
- **pooltype** (*str*) – Pooling type. It must be either `'max'` or `'avg'`.
- **proj_pool** (*int or None*) – Projection size in the pooling path. If `None`, no projection is done.
- **stride** (*int*) – Stride parameter of the last convolution of each path.
- **conv_init** (*initializer*) – Initializer to initialize the convolution matrix weights. When it is `numpy.ndarray`, its `ndim` should be 4.
- **dtype** (*numpy.dtype*) – Type to use in *BatchNormalization*.

See also:

Inception

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link.

`copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (AbstractSerializer) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override device_resident_accept() to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Linear

class chainer.links.**Linear**(*in_size, out_size=None, nobias=False, initialW=None, initial_bias=None*)
Linear layer (a.k.a. fully-connected layer).

This is a link that wraps the *linear()* function, and holds a weight matrix `W` and optionally a bias vector `b` as parameters.

If `initialW` is left to the default value of `None`, the weight matrix `W` is initialized with i.i.d. Gaussian samples, each of which has zero mean and deviation $\sqrt{1/}$

Parameters
- **in_size** (*int or None*) – Dimension of input vectors. If unspecified or `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (*int*) – Dimension of output vectors. If only one value is passed for `in_size` and `out_size`, that value will be used for the `out_size` dimension.
- **nobias** (*bool*) – If `True`, then this function does not use the bias.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 2. If `initialW` is `None`, then the weights are initialized with i.i.d. Gaussian samples, each of which has zero mean and deviation $\sqrt{1/}$

See also:

linear()

- **Variables**
  - **W** (`Variable`) – Weight parameter.
  - **b** (`Variable`) – Bias parameter.

Example

There are several ways to make a Linear link.

Define an input vector `x` as:

```
>>> x = np.array([[0, 1, 2, 3, 4]], np.float32)
```

1. Give the first two arguments explicitly:

   Those numbers are considered as the input size and the output size.

   ```
   >>> l = L.Linear(5, 10)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

2. Omit `in_size` (give the output size only as the first argument) or fill it with `None`:

   In this case, the size of second axis of `x` is used as the input size. So the below two cases are the same.

   ```
   >>> l = L.Linear(10)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   ```
   >>> l = L.Linear(None, 10)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   When you omit the first argument, you need to specify the other subsequent arguments from `nobias` as keyword arguments. So the below two cases are the same.

   ```
   >>> l = L.Linear(None, 10, False, None, 0)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   ```
   >>> l = L.Linear(10, nobias=False, initialW=None, initial_bias=0)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*, *n_batch_axes=1*)
Applies the linear layer.

Parameters

- **x** (`Variable`) – Batch of input vectors.
- **n_batch_axes** (*int*) – The number of batch axes. The default is 1. The input variable is reshaped into $(\text{n\_batch\_axes} + 1)$-dimensional tensor. This should be greater than 0.

Returns Output of the linear layer.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See *`init_scope()`* for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *`chainerx`*.

chainer.links.LocalConvolution2D

class `chainer.links.`**LocalConvolution2D**(*in_channels*, *out_channels*, *in_size=None*, *ksize=None*, *stride=1*, *nobias=False*, *initialW=None*, *initial_bias=None*, ***kwargs*)

Two-dimensional local convolutional layer.

This link wraps the *`local_convolution_2d()`* function and holds the filter weight and bias array as parameters.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays. If either in_channels or in_size is `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*int*) – Number of channels of output arrays
- **in_size** (*int or pair of ints*) – Size of each image channel `in_size=k` and `in_size=(k,k)` are equivalent. If either in_channels or in_size is `None`, parameter initialization will be deferred until the first forward data pass when the size will be determined.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **nobias** (*bool*) – If `True`, then this link does not use the bias term.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 6.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 3.

See also:

See `chainer.functions.local_convolution_2d()`.

Variables

- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

`addgrads`(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*x*)
Applies the local convolution layer.

Parameters x (Variable) – Input image.

Returns Output of the convolution.

Return type *Variable*

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a Sequential.

This method returns a Sequential object which has the same Link multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer Sequential block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.

- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.LSTM

class chainer.links.**LSTM**(*in_size*, *out_size=None*, *lateral_init=None*, *upward_init=None*, *bias_init=None*, *forget_bias_init=None*)
Fully-connected LSTM layer.

This is a fully-connected LSTM layer as a chain. Unlike the `lstm()` function, which is defined as a stateless activation function, this chain holds upward and lateral connections as child links.

It also maintains *states*, including the cell state and the output at the previous time step. Therefore, it can be used as a *stateful LSTM*.

This link supports variable length inputs. The mini-batch size of the current input must be equal to or smaller than that of the previous one. The mini-batch size of `c` and `h` is determined as that of the first input `x`. When mini-batch size of `i`-th input is smaller than that of the previous input, this link only updates `c[0:len(x)]` and `h[0:len(x)]` and doesn't change the rest of `c` and `h`. So, please sort input sequences in descending order of lengths before applying the function.

Parameters

- **in_size** (*int*) – Dimension of input vectors. If it is `None` or omitted, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (*int*) – Dimensionality of output vectors.
- **lateral_init** – A callable that takes *N-dimensional array* and edits its value. It is used for initialization of the lateral connections. May be `None` to use default initialization.
- **upward_init** – A callable that takes *N-dimensional array* and edits its value. It is used for initialization of the upward connections. May be `None` to use default initialization.
- **bias_init** – A callable that takes *N-dimensional array* and edits its value It is used for initialization of the biases of cell input, input gate and output gate.and gates of the upward connection. May be a scalar, in that case, the bias is initialized by this value. If it is `None`, the cell-input bias is initialized to zero.
- **forget_bias_init** – A callable that takes *N-dimensional array* and edits its value. It is used for initialization of the biases of the forget gate of the upward connection. May be a scalar, in that case, the bias is initialized by this value. If it is `None`, the forget bias is initialized to one.

Variables

- **upward** (`Linear`) – Linear layer of upward connections.
- **lateral** (`Linear`) – Linear layer of lateral connections.
- **c** (`Variable`) – Cell states of LSTM units.
- **h** (`Variable`) – Output at the previous time step.

Example

There are several ways to make a LSTM link.

Let a two-dimensional input array x be:

```
>>> x = np.zeros((1, 10), dtype=np.float32)
```

1. Give both `in_size` and `out_size` arguments:

```
>>> l = L.LSTM(10, 20)
>>> h_new = l(x)
>>> h_new.shape
(1, 20)
```

2. Omit `in_size` argument or fill it with `None`:

The below two cases are the same.

```
>>> l = L.LSTM(20)
>>> h_new = l(x)
>>> h_new.shape
(1, 20)
```

```
>>> l = L.LSTM(None, 20)
>>> h_new = l(x)
>>> h_new.shape
(1, 20)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

forward(*x*)
Updates the internal state and returns the LSTM outputs.

Parameters **x** (Variable) – A new batch from the input sequence.

Returns Outputs of updated LSTM units.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

Resets the internal state.

It sets `None` to the `c` and `h` attributes.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

set_state(*c*, *h*)

Sets the internal state.

It sets the `c` and `h` attributes.

Parameters

- **c** (`Variable`) – A new cell states of LSTM units.
- **h** (`Variable`) – A new output at the previous time step.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.MLPConvolution2D

class chainer.links.**MLPConvolution2D**(*self*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *activation=relu.relu*, *conv_init=None*, *bias_init=None*)

Two-dimensional MLP convolution layer of Network in Network.

This is an "mlpconv" layer from the Network in Network paper. This layer is a two-dimensional convolution layer followed by 1x1 convolution layers and interleaved activation functions.

Note that it does not apply the activation function to the output of the last 1x1 convolution layer.

Parameters

- **in_channels** (*int or None*) – Number of channels of input arrays. If it is `None` or omitted, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** (*tuple of ints*) – Tuple of number of channels. The i-th integer indicates the number of filters of the i-th convolution.
- **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels) of the first convolution layer. `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or pair of ints*) – Stride of filter applications at the first convolution layer. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or pair of ints*) – Spatial padding width for input arrays at the first convolution layer. `pad=p` and `pad=(p, p)` are equivalent.

- **activation** (*callable*) – Activation function for internal hidden units. You can specify one of activation functions from *built-in activation functions* or your own function. It should not be an *activation functions with parameters* (i.e., *Link* instance). The function must accept one argument (the output from each child link), and return a value. Returned value must be a Variable derived from the input Variable to perform backpropagation on the variable. Note that this function is not applied to the output of this link.
- **conv_init** – An initializer of weight matrices passed to the convolution layers. This option must be specified as a keyword argument.
- **bias_init** – An initializer of bias vectors passed to the convolution layers. This option must be specified as a keyword argument.

See: Network in Network.

Variables **activation** (*callable*) – Activation function. See the description in the arguments for details.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

append (*value*)
S.append(value) – append value to the end of the sequence

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear () → None – remove all items from S

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Returns a deep copy of the chainlist.

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using copy.deepcopy(). The old behavior (not copying persistent values) can be reproduced with copy_persistent=False.

Parameters

- **link** (Link) – Source link object.

- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count (*value*) → integer – return number of occurrences of value

count_params ()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend (*values*)

S.extend(iterable) – extend sequence by appending elements from the iterable

forward (*x*)

Computes the output of the mlpconv layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of the mlpconv layer.

Return type *Variable*

from_chx ()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

init_scope ()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

insert (*index, link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (`int`) – Number of times to repeat.
- **mode** (`str`) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse ()
S.reverse() – reverse *IN PLACE*

serialize (*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.NaryTreeLSTM

class chainer.links.**NaryTreeLSTM**(*in_size*, *out_size*, *n_ary=2*)
N-ary TreeLSTM unit.

> **Warning:** This feature is experimental. The interface can change in the future.

This is a N-ary TreeLSTM unit as a chain. This link is a fixed-length arguments function, which compounds the states of all children nodes into the new states of a current (parent) node. *states* denotes the cell state, c, and the output, h, which are produced by this link. This link doesn't keep cell and hidden states internally.

For example, this link is called such as `func(c1, c2, h1, h2, x)` if the number of children nodes was set 2 (`n_ary = 2`), while `func(c1, c2, c3, h1, h2, h3, x)` if that was 3 (`n_ary = 3`). This function is *dependent* from an order of children nodes unlike Child-Sum TreeLSTM. Thus, the returns of `func(c1, c2, h1, h2, x)` are different from those of `func(c2, c1, h2, h1, x)`.

Parameters

- **in_size** (`int`) – Dimension of input vectors.
- **out_size** (`int`) – Dimensionality of cell and output vectors.
- **n_ary** (`int`) – The number of children nodes in a tree structure.

Variables

- **W_x** (`chainer.links.Linear`) – Linear layer of connections from input vectors.
- **W_h** (`chainer.links.Linear`) – Linear layer of connections between (a, i, o, all f) and the output of each child. a, i, o and f denotes input compound, input gate, output gate and forget gate, respectively. a, input compound, equals to u in the paper by Tai et al.

See the papers for details: Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks, and A Fast Unified Model for Parsing and Sentence Understanding.

Tai et al.'s N-Ary TreeLSTM is little extended in Bowman et al., and this link is based on the variant by Bowman et al. Specifically, eq. 10 in Tai et al. has only one W matrix to be applied to x, consistently for all children. On the other hand, Bowman et al.'s model has multiple matrices, each of which affects the forget gate for each child's cell individually.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link, copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.

- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**cshsx*)
Returns new cell state and output of N-ary TreeLSTM.

Parameters **cshsx** (list of *Variable*) – Arguments which include all cell vectors and all output vectors of fixed-length children, and an input vector. The number of arguments must be same as `n_ary * 2 + 1`.

Returns Returns (c_{new}, h_{new}), where c_{new} represents new cell state vector, and h_{new} is new output vector.

Return type tuple of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NStepBiGRU

class chainer.links.**NStepBiGRU**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)
Stacked Bi-directional GRU for sequences.

This link is stacked version of Bi-directional GRU for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike *chainer.functions.n_step_bigru()*, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of *chainer.Variable* holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_bigru()

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (`Variable` or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of `Variable`) – List of input sequences. Each element `xs[i]` is a `chainer.Variable` holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of `Variable` . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert(*index*, *link*)
Insert a child link at the given index.

Parameters

- **index** (`int`) – The position of the list where the new
- **is inserted.** (`link`) –
- **link** (`Link`) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a Sequential.

This method returns a Sequential object which has the same Link multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer Sequential block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
```

(continues on next page)

(continued from previous page)

```
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()

S.reverse() – reverse *IN PLACE*

rnn(**args*)

Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 6

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = True

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.NStepBiLSTM

class chainer.links.**NStepBiLSTM**(*self, n_layers, in_size, out_size, dropout*)
Stacked Bi-directional LSTM for sequences.

This link is stacked version of Bi-directional LSTM for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_bilstm()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

Parameters

- **n_layers** (`int`) – Number of layers.
- **in_size** (`int`) – Dimensionality of input vectors.
- **out_size** (`int`) – Dimensionality of hidden states and output vectors.
- **dropout** (`float`) – Dropout ratio.

See also:

`chainer.functions.n_step_bilstm()`

Methods

__call__(**args, **kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()

Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)

Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)

Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *cx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional LSTM and `(2S, B, N)` for bi-directional LSTM where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **cx** (*Variable* or None) – Initial cell states. If `None` is specified zero-vector is used. It has the same shape as `hx`.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_i` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing three elements, `hy`, `cy` and `ys`.

- `hy` is an updated hidden states whose shape is the same as `hx`.
- `cy` is an updated cell states whose shape is the same as `cx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional LSTM and `(L_i, 2N)` for bi-directional LSTM where `L_i` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

insert (*index*, *link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat(*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()

S.reverse() – reverse *IN PLACE*

rnn(**args*)

Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 8

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = True

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NStepBiRNNReLU

class chainer.links.**NStepBiRNNReLU**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)
Stacked Bi-directional RNN for sequences.

This link is stacked version of Bi-directional RNN for sequences. Note that the activation function is `relu`. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike *chainer.functions.n_step_birnn()*, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of *chainer.Variable* holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.

- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_birnn()

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

`__setitem__`(*index*, *value*)

`__len__`()
Returns the number of children.

`__iter__`()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append (*value*)
S.append(value) – append value to the end of the sequence

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear () → None – remove all items from S

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Returns a deep copy of the chainlist.

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count (*value*) → integer – return number of occurrences of value

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

insert (*index, link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters skipself (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters include_uninit (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters include_uninit (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse ()
S.reverse() – reverse *IN PLACE*

rnn(**args*)
Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 2

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = True

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NStepBiRNNTanh

class chainer.links.**NStepBiRNNTanh**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)
Stacked Bi-directional RNN for sequences.

This link is stacked version of Bi-directional RNN for sequences. Note that the activation function is `tanh`. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_birnn()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

`chainer.functions.n_step_birnn()`

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

`__setitem__`(*index*, *value*)

`__len__`()
Returns the number of children.

`__iter__`()

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

append (*value*)
S.append(value) – append value to the end of the sequence

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear () → None – remove all items from S

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Returns a deep copy of the chainlist.

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of BatchNormalization). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using copy.deepcopy(). The old behavior (not copying persistent values) can be reproduced with copy_persistent=False.

Parameters

- **link** (Link) – Source link object.

- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count (*value*) → integer – return number of occurrences of value

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend (*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward (*self*, *hx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

insert(*index*, *link*)
Insert a child link at the given index.

Parameters

- **index** (`int`) – The position of the list where the new
- **is inserted.** (`link`) –
- **link** (`Link`) – The link to be inserted.

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial

parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()
S.reverse() – reverse *IN PLACE*

rnn(**args*)
Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 2

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = True

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *`chainerx`*.

chainer.links.NStepGRU

class `chainer.links.`**NStepGRU**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)

Stacked Uni-directional GRU for sequences.

This link is stacked version of Uni-directional GRU for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike *`chainer.functions.n_step_gru()`*, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of *`chainer.Variable`* holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

`chainer.functions.n_step_gru()`

Methods

__call__(**args*, ***kwargs*)

Call self as a function.

__getitem__(*index*)

Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()

Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)

S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *xs*)

Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable*. Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert(*index*, *link*)

Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()

S.reverse() – reverse *IN PLACE*

rnn(**args*)

Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

`Device` instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells

Returns the number of cells.

This function must be implemented in a child class.

n_weights = 6

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = False

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NStepLSTM

class chainer.links.**NStepLSTM**(*self, n_layers, in_size, out_size, dropout*)
Stacked Uni-directional LSTM for sequences.

This link is stacked version of Uni-directional LSTM for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike *chainer.functions.n_step_lstm()*, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of *chainer.Variable* holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_lstm()

Methods

__call__(**args, **kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index, value*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook, name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *cx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional LSTM and `(2S, B, N)` for bi-directional

LSTM where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.

- **cx** (*Variable* or None) – Initial cell states. If `None` is specified zero-vector is used. It has the same shape as `hx`.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_i` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing three elements, `hy`, `cy` and `ys`.

- `hy` is an updated hidden states whose shape is the same as `hx`.
- `cy` is an updated cell states whose shape is the same as `cx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional LSTM and `(L_i, 2N)` for bi-directional LSTM where `L_i` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert(*index*, *link*)

Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –

- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
```

(continues on next page)

(continued from previous page)

```
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()

S.reverse() – reverse *IN PLACE*

rnn(**args*)

Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 8

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = False

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NStepRNNReLU

class chainer.links.**NStepRNNReLU**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)
Stacked Uni-directional RNN for sequences.

This link is stacked version of Uni-directional RNN for sequences. Note that the activation function is `relu`. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike *chainer.functions.n_step_rnn()*, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of *chainer.Variable* holding sequences.

Parameters

- **n_layers** (*int*) – Number of layers.
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_rnn()

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

`__setitem__`(*index*, *value*)

`__len__`()
Returns the number of children.

`__iter__`()

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward(*self*, *hx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx(*xs*)

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

insert (*index*, *link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse ()
S.reverse() – reverse *IN PLACE*

rnn (**args*)
Calls RNN function.

This function must be implemented in a child class.

serialize (*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 2

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = False

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.NStepRNNTanh

class `chainer.links.`**NStepRNNTanh**(*self*, *n_layers*, *in_size*, *out_size*, *dropout*)
Stacked Uni-directional RNN for sequences.

This link is stacked version of Uni-directional RNN for sequences. Note that the activation function is `tanh`. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_rnn()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

Parameters

- **n_layers** (`int`) – Number of layers.

- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of hidden states and output vectors.
- **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_rnn()

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append (*value*)
S.append(value) – append value to the end of the sequence

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear () → None – remove all items from S

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Returns a deep copy of the chainlist.

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count (*value*) → integer – return number of occurrences of value

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend (*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

forward (*self*, *hx*, *xs*)
Calculates all of the hidden states and the cell states.

Parameters

- **hx** (*Variable* or None) – Initial hidden states. If `None` is specified zero-vector is used. Its shape is `(S, B, N)` for uni-directional RNN and `(2S, B, N)` for bi-directional RNN where `S` is the number of layers and is equal to `n_layers`, `B` is the mini-batch size, and `N` is the dimension of the hidden units.
- **xs** (list of *Variable*) – List of input sequences. Each element `xs[i]` is a *chainer.Variable* holding a sequence. Its shape is `(L_i, I)`, where `L_t` is the length of a sequence for batch `i`, and `I` is the size of the input and is equal to `in_size`.

Returns

This function returns a tuple containing two elements, `hy` and `ys`.

- `hy` is an updated hidden states whose shape is same as `hx`.
- `ys` is a list of *Variable* . Each element `ys[i]` holds hidden states of the last layer corresponding to an input `xs[i]`. Its shape is `(L_i, N)` for uni-directional RNN and `(L_i, 2N)` for bi-directional RNN where `L_t` is the length of a sequence for batch `i`, and `N` is size of hidden units.

Return type tuple

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_hx (*xs*)

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert (*index, link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (Link) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse ()
S.reverse() – reverse *IN PLACE*

rnn(**args*)
Calls RNN function.

This function must be implemented in a child class.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

n_cells
Returns the number of cells.

This function must be implemented in a child class.

n_weights = 2

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

use_bi_direction = False

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Parameter

class chainer.links.**Parameter**(*array*)
Link that just holds a parameter and returns it.

Deprecated since version v1.5: The parameters are stored as variables since v1.5. Use them directly instead.

Parameters **array** – Initial parameter array.

Variables **W** (Variable) – Parameter variable.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*volatile='off'*)
Returns the parameter variable.

Parameters **volatile** (*Flag*) – The volatility of the returned variable.

Returns A copy of the parameter variable with given volatility.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial

parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device (*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu (*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64 ()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds ()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Scale

class chainer.links.**Scale**(*axis=1*, *W_shape=None*, *bias_term=False*, *bias_shape=None*)
Broadcasted elementwise product with learnable parameters.

Computes a elementwise product as *scale()* function does except that its second input is a learnable weight parameter W the link has.

Parameters

- **axis** (*int*) – The first axis of the first input of *scale()* function along which its second input is applied.

- **W_shape** (*tuple of ints*) – Shape of learnable weight parameter. If None, this link does not have learnable weight parameter so an explicit weight needs to be given to its forward method's second input.
- **bias_term** (*bool*) – Whether to also learn a bias (equivalent to Scale link + Bias link).
- **bias_shape** (*tuple of ints*) – Shape of learnable bias. If W_shape is None, this should be given to determine the shape. Otherwise, the bias has the same shape W_shape with the weight parameter and bias_shape is ignored.

See also:

See *scale()* for details.

Variables

- **W** (Parameter) – Weight parameter if W_shape is given. Otherwise, no W attribute.
- **bias** (Bias) – Bias term if bias_term is True. Otherwise, no bias attribute.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a

scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise,

it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**xs*)

Applies broadcasted elementwise product.

Parameters **xs** (*list of Variables*) – Input variables whose length should be one if the link has a learnable weight parameter, otherwise should be two.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

`Device` instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.StatefulGRU

class `chainer.links.StatefulGRU`(*in_size*, *out_size*, *init=None*, *inner_init=None*, *bias_init=0*)

Stateful Gated Recurrent Unit function (GRU).

Stateful GRU function has six parameters W_r, W_z, W, U_r, U_z, and U. The three parameters W_r, W_z, and W are $n \times m$ matrices, and the others U_r, U_z, and U are $n \times n$ matrices, where m is the length of input vectors and n is the length of hidden vectors.

Given input vector x, Stateful GRU returns the next hidden vector h' defined as

$$
\begin{aligned}
r &= \\
\sigma(W_r x + U_r h), \\
z &= \\
\sigma(W_z x + U_z h), \\
\bar{h} &= \\
\tanh(Wx + U(r \odot h)), \\
h' &= \\
(1 - z) \odot h + z \odot \bar{h},
\end{aligned}
$$

where h is current hidden vector.

As the name indicates, `StatefulGRU` is *stateful*, meaning that it also holds the next hidden vector *h'* as a state. For a *stateless* GRU, use `StatelessGRU`.

Parameters

- **in_size** (*int*) – Dimension of input vector x.
- **out_size** (*int*) – Dimension of hidden vector h.
- **init** – Initializer for GRU's input units (W). It is a callable that takes *N-dimensional array* and edits its value. If it is `None`, the default initializer is used.
- **inner_init** – Initializer for the GRU's inner recurrent units (U). It is a callable that takes *N-dimensional array* and edits its value. If it is `None`, the default initializer is used.
- **bias_init** – Bias initializer. It is a callable that takes *N-dimensional array* and edits its value. If `None`, the bias is set to zero.

Variables **h** (`Variable`) – Hidden vector that indicates the state of `StatefulGRU`.

See also:

- *StatelessGRU*
- *GRU*: an alias of *StatefulGRU*

Example

There are several ways to make a `StatefulGRU` link. Let `x` be a two-dimensional input array:

```
>>> in_size = 10
>>> out_size = 20
>>> x = np.zeros((1, in_size), dtype=np.float32)
```

1. Give only `in_size` and `out_size` arguments:

```
>>> l = L.StatefulGRU(in_size, out_size)
>>> h_new = l(x)
>>> h_new.shape
(1, 20)
```

2. Give all optional arguments:

```
>>> init = np.zeros((out_size, in_size), dtype=np.float32)
>>> inner_init = np.zeros((out_size, out_size), dtype=np.float32)
>>> bias = np.zeros((1, out_size), dtype=np.float32)
>>> l = L.StatefulGRU(in_size, out_size, init=init,
...     inner_init=inner_init, bias_init=bias)
>>> h_new = l(x)
>>> h_new.shape
(1, 20)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

set_state(*h*)

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

`to_device`(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **`device`** – Target device specifier. See *`get_device()`* for available values.

Returns: self

`to_gpu`(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **`device`** – Target device specifier. If omitted, the current device is used.

Returns: self

`to_intel64`()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

`zerograds`()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *`cleargrads()`* instead.

`__eq__`()

Return self==value.

`__ne__`()

Return self!=value.

`__lt__`()

Return self<value.

`__le__`()

Return self<=value.

`__gt__`()

Return self>value.

`__ge__`()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.StatelessGRU

class `chainer.links.`**StatelessGRU**(*in_size*, *out_size*, *init=None*, *inner_init=None*, *bias_init=None*)

Stateless Gated Recurrent Unit function (GRU).

GRU function has six parameters W_r, W_z, W, U_r, U_z, and U. The three parameters W_r, W_z, and W are $n \times m$ matrices, and the others U_r, U_z, and U are $n \times n$ matrices, where m is the length of input vectors and n is the length of hidden vectors.

Given two inputs a previous hidden vector h and an input vector x, GRU returns the next hidden vector h' defined as

$$r = \sigma(W_r x + U_r h),$$
$$z = \sigma(W_z x + U_z h),$$
$$\bar{h} = \tanh(W x + U(r \odot h)),$$
$$h' = (1 - z) \odot h + z \odot \bar{h},$$

where σ is the sigmoid function, and $\odot$ is the element-wise product.

As the name indicates, `StatelessGRU` is *stateless*, meaning that it does not hold the value of hidden vector h. For a *stateful* GRU, use `StatefulGRU`.

Parameters

- **in_size** (*int*) – Dimension of input vector x. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (*int*) – Dimension of hidden vector h, $\bar{h}$ and h'.

See:

- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches [Cho+, SSST2014].
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling [Chung+NIPS2014 DLWorkshop].

See also:

StatefulGRU

Example

There are several ways to make a StatelessGRU link. Let x be a two-dimensional input array:

```
>>> in_size = 10
>>> out_size = 20
>>> x = np.zeros((1, in_size), dtype=np.float32)
>>> h = np.zeros((1, out_size), dtype=np.float32)
```

1. Give both in_size and out_size arguments:

```
>>> l = L.StatelessGRU(in_size, out_size)
>>> h_new = l(h, x)
>>> h_new.shape
(1, 20)
```

2. Omit in_size argument or fill it with None:

```
>>> l = L.StatelessGRU(None, out_size)
>>> h_new = l(h, x)
>>> h_new.shape
(1, 20)
```

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link (*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*h*, *x*)

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.StatefulMGU

class chainer.links.**StatefulMGU**(*in_size*, *out_size*)

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.

- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters link (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters mode (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters name (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

set_state(*h*)

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.StatelessMGU

class chainer.links.**StatelessMGU**(*n_inputs*, *n_units*)

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.

- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (`bool`) – If `True`, persistent values are also copied. `True` by default.

count_params ()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)

Unregisters the link hook.

Parameters **name** (`str`) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*h*, *x*)

from_chx ()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.StatefulPeepholeLSTM

class `chainer.links.StatefulPeepholeLSTM`(*in_size*, *out_size*)
Fully-connected LSTM layer with peephole connections.

This is a fully-connected LSTM layer with peephole connections as a chain. Unlike the `LSTM` link, this chain holds `peep_i`, `peep_f` and `peep_o` as child links besides `upward` and `lateral`.

Given a input vector x, Peephole returns the next hidden vector h' defined as

$$
\begin{aligned}
a &= \\
&\tanh(upward x + lateral h), \\
i &= \\
&\sigma(upward x + lateral h + peep_i c), \\
f &= \\
&\sigma(upward x + lateral h + peep_f c), \\
c' &= \\
&a \odot i + f \odot c, \\
o &= \\
&\sigma(upward x + lateral h + peep_o c'), \\
h' &= \\
&o \tanh(c'),
\end{aligned}
$$

where σ is the sigmoid function, $\odot$ is the element-wise product, c is the current cell state, c' is the next cell state and h is the current hidden vector.

Parameters

- **in_size** (*int*) – Dimension of the input vector x.
- **out_size** (*int*) – Dimension of the hidden vector h.

Variables

- **upward** (Linear) – Linear layer of upward connections.
- **lateral** (Linear) – Linear layer of lateral connections.
- **peep_i** (Linear) – Linear layer of peephole connections to the input gate.
- **peep_f** (Linear) – Linear layer of peephole connections to the forget gate.
- **peep_o** (Linear) – Linear layer of peephole connections to the output gate.
- **c** (Variable) – Cell states of LSTM units.
- **h** (Variable) – Output at the current time step.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

forward(*x*)
Updates the internal state and returns the LSTM outputs.

Parameters **x** (Variable) – A new batch from the input sequence.

Returns Outputs of updated LSTM units.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

Resets the internal states.

It sets `None` to the `c` and `h` attributes.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.StatefulZoneoutLSTM

class `chainer.links.`**StatefulZoneoutLSTM**(*in_size*, *out_size*, *c_ratio=0.5*, *h_ratio=0.5*, ***kwargs*)

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (`str`) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (`str`) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (`str`) – Name of the parameter. This name is also used as the attribute name.
- **shape** (`int or tuple of ints`) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)

Updates the internal state and returns the LSTM outputs.

Parameters **x** (`Variable`) – A new batch from the input sequence.

Returns Outputs of updated LSTM units.

Return type *Variable*

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reset_state()

Resets the internal state.

It sets `None` to the `c` and `h` attributes.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

set_state(*c*, *h*)

Sets the internal state.

It sets the `c` and `h` attributes.

Parameters

- **c** (`Variable`) – A new cell states of LSTM units.
- **h** (`Variable`) – A new output at the previous time step.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.StatelessLSTM

class chainer.links.**StatelessLSTM**(*in_size*, *out_size=None*, *lateral_init=None*, *upward_init=None*, *bias_init=None*, *forget_bias_init=None*)
Stateless LSTM layer.

This is a fully-connected LSTM layer as a chain. Unlike the `lstm()` function, this chain holds upward and lateral connections as child links. This link doesn't keep cell and hidden states.

Parameters

- **in_size** (*int or None*) – Dimension of input vectors. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (*int*) – Dimensionality of output vectors.

Variables

- **upward** (`chainer.links.Linear`) – Linear layer of upward connections.
- **lateral** (`chainer.links.Linear`) – Linear layer of lateral connections.

Example

There are several ways to make a StatelessLSTM link.

Let a two-dimensional input array x, a cell state array h, and the output array of the previous step h be:

```
>>> x = np.zeros((1, 10), dtype=np.float32)
>>> c = np.zeros((1, 20), dtype=np.float32)
>>> h = np.zeros((1, 20), dtype=np.float32)
```

1. Give both `in_size` and `out_size` arguments:

```
>>> l = L.StatelessLSTM(10, 20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

2. Omit `in_size` argument or fill it with `None`:

   The below two cases are the same.

```
>>> l = L.StatelessLSTM(20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

```
>>> l = L.StatelessLSTM(None, 20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.

- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters name (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*c*, *h*, *x*)
Returns new cell state and updated output of LSTM.

Parameters

- **c** (`Variable`) – Cell states of LSTM units.
- **h** (`Variable`) – Output at the previous time step.
- **x** (`Variable`) – A new batch from the input sequence.

Returns Returns `(c_new, h_new)`, where `c_new` represents new cell state, and `h_new` is updated output of LSTM units.

Return type tuple of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.3.2 Activation/loss/normalization functions with parameters

`chainer.links.BatchNormalization`	Batch normalization layer on outputs of linear or convolution functions.
`chainer.links.BatchRenormalization`	Batch renormalization layer on outputs of linear or convolution functions.
`chainer.links.DecorrelatedBatchNormalization`	Decorrelated batch normalization layer.
`chainer.links.GroupNormalization`	Group normalization layer on outputs of convolution functions.
`chainer.links.LayerNormalization`	Layer normalization layer on outputs of linear functions.
`chainer.links.BinaryHierarchicalSoftmax`	Hierarchical softmax layer over binary tree.
`chainer.links.BlackOut`	BlackOut loss layer.
`chainer.links.CRF1d`	Linear-chain conditional random field loss layer.
`chainer.links.SimplifiedDropconnect`	Fully-connected layer with simplified dropconnect regularization.
`chainer.links.PReLU`	Parametric ReLU function as a link.
`chainer.links.Swish`	Swish activation function as a link.
`chainer.links.Maxout`	Fully-connected maxout layer.
`chainer.links.NegativeSampling`	Negative sampling loss layer.

chainer.links.BatchNormalization

class chainer.links.**BatchNormalization**(*size=None*, *decay=0.9*, *eps=2e-05*, *dtype=None*, *use_gamma=True*, *use_beta=True*, *initial_gamma=None*, *initial_beta=None*, *axis=None*, *initial_avg_mean=None*, *initial_avg_var=None*)

Batch normalization layer on outputs of linear or convolution functions.

This link wraps the `batch_normalization()` and `fixed_batch_normalization()` functions.

It runs in three modes: training mode, fine-tuning mode, and testing mode.

In training mode, it normalizes the input by *batch statistics*. It also maintains approximated population statistics by moving averages, which can be used for instant evaluation in testing mode. Training mode is enabled when `chainer.config.train` is set to `True` and `__call__()` is invoked with `finetune=False` (the default is False).

In fine-tuning mode, it accumulates the input to compute *population statistics*. In order to correctly compute the population statistics, a user must use this mode to feed mini-batches running through whole training dataset. Finetuning mode is enabled when `chainer.config.train` is set to `True` and `__call__()` is invoked with `finetune=True`.

In testing mode, it uses pre-computed population statistics to normalize the input variable. The population statistics is approximated if it is computed by training mode, or accurate if it is correctly computed by fine-tuning mode. Testing mode is enabled when `chainer.config.train` is set to `False`.

Parameters

- **size** (*int, tuple of ints, or None*) – Size (or shape) of channel dimensions. If `None`, the size will be determined from dimension(s) of the input batch during the first forward pass.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability.
- **dtype** (*numpy.dtype*) – Type to use in computing.
- **use_gamma** (*bool*) – If `True`, use scaling parameter. Otherwise, use unit(1) which makes no effect.
- **use_beta** (*bool*) – If `True`, use shifting parameter. Otherwise, use unit(0) which makes no effect.
- **axis** (*int or tuple of int*) – Axis over which normalization is performed. When axis is `None`, it is determined from input dimensions. For example, if `x.ndim` is 4, axis becomes (0, 2, 3) and normalization is performed over 0th, 2nd and 3rd axis of input. If it is 2, axis becomes (0) and normalization is performed over 0th axis of input. When a tuple of int is given to this option, numbers in the tuple must be being sorted in ascending order. For example, (0, 2) is OK, but (2, 0) is not.
- **initial_gamma** – Initializer of the scaling parameter. The default value is `1`.
- **initial_beta** – Initializer of the shifting parameter. The default value is `0`.
- **initial_avg_mean** – Initializer of the moving average of population mean. The default value is `0`.
- **initial_avg_var** – Initializer of the moving average of population variance. The default value is `1`.

Note: From v5.0.0, the initial value of the population variance is changed to 1. It does not change the behavior of training, but the resulting model may have a slightly different behavior on inference. To emulate the old behavior, pass `initial_avg_var=0` for training.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

See also:

batch_normalization(), *fixed_batch_normalization()*

Variables

- **gamma** (`Variable`) – Scaling parameter. In mixed16 mode, it is initialized as float32 variable.
- **beta** (`Variable`) – Shifting parameter. In mixed16 mode, it is initialized as float32 variable.
- **avg_mean** (*N-dimensional array*) – Population mean. In mixed16 mode, it is initialized as float32 array.
- **avg_var** (*N-dimensional array*) – Population variance. In mixed16 mode, it is initialized as float32 array.
- **N** (*int*) – Count of batches given for fine-tuning.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability. This value is added to the batch variances.

Example

```
>>> x = np.arange(12).reshape(4, 3).astype(np.float32) ** 2
>>> x
array([[  0.,   1.,   4.],
       [  9.,  16.,  25.],
       [ 36.,  49.,  64.],
       [ 81., 100., 121.]], dtype=float32)
>>> bn = chainer.links.BatchNormalization(3)
>>> bn(x)
variable([[-1.        , -1.0664359 , -1.1117983 ],
          [-0.71428573, -0.6714596 , -0.6401263 ],
          [ 0.14285715,  0.19748813,  0.23583598],
          [ 1.5714287 ,  1.5404074 ,  1.5160885 ]])
>>> (x - x.mean(axis=0)) / np.sqrt(x.var(axis=0) + 2e-5)
array([[-1.        , -1.0664359 , -1.1117983 ],
       [-0.71428573, -0.6714596 , -0.6401263 ],
       [ 0.14285715,  0.19748813,  0.235836  ],
       [ 1.5714285 ,  1.5404074 ,  1.5160886 ]], dtype=float32)
```

There are several ways to make a BatchNormalization link. Consider an input of batched 10 images of 32x32 with 3 channels.

```
>>> x = np.random.randn(10, 3, 32, 32).astype(np.float32)
```

1. Give the parameter size:

To normalize for each channel, give the number of channels to `size`.

```
>>> bn = chainer.links.BatchNormalization(3)
>>> bn.avg_mean.shape
(3,)
>>> bn.beta += 2.0
>>> bn.gamma *= 5.0
>>> list(sorted(bn.namedparams()))
[('/beta', variable([2., ...])), ('/gamma', variable([5., ...]))]
>>> y = bn(x)
>>> y.shape
(10, 3, 32, 32)
>>> np.testing.assert_allclose(
...     y.array.mean(axis=(0, 2, 3)), bn.beta.array, atol=1e-6)
>>> np.testing.assert_allclose(
...     y.array.std(axis=(0, 2, 3)),
...     bn.gamma.array, atol=1e-3)
```

To normalize for each channel for each pixel, `size` should be the tuple of the dimensions.

```
>>> bn = chainer.links.BatchNormalization((3, 32, 32))
>>> bn.avg_mean.shape
(3, 32, 32)
>>> y = bn(x)
>>> y.shape
(10, 3, 32, 32)
>>> np.testing.assert_allclose(
...     y.array.mean(axis=0), bn.beta.array, atol=1e-6)
>>> np.testing.assert_allclose(
...     y.array.std(axis=0),
...     bn.gamma.array, atol=1e-3)
```

By default, channel axis is (or starts from) the 1st axis of the input shape.

2. Give the aggregate axes:

   from Chainer v5

   With `axis` option, similarly to NumPy, you may specify the aggregate axes, which are treated as the "batch" axes for the batch statistics.

   You can omit `size` if `axis` is given. In this case, creation of persistent values `avg_mean`, `avg_var` and parameters `beta`, `gamma` is deferred until first forward propagation.

   The examples in 1. corresponds to the following, respectively.

```
>>> bn = chainer.links.BatchNormalization(axis=(0, 2, 3))
>>> print(bn.avg_mean)
None
>>> y = bn(x)
>>> bn.avg_mean.shape
(3,)
```

```
>>> bn = chainer.links.BatchNormalization(axis=0)
>>> print(bn.avg_mean)
None
>>> y = bn(x)
>>> bn.avg_mean.shape
(3, 32, 32)
```

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*`str`*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*`int or tuple of ints`*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*`str`*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

`addgrads`(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

`children`()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*self*, *x*, *finetune=False*)
Invokes the forward propagation of BatchNormalization.

In training mode, the BatchNormalization computes moving averages of mean and variance for evaluation during training, and normalizes the input using batch statistics.

Parameters

- **x** (`Variable`) – Input variable.
- **finetune** (*bool*) – If it is in the training mode and `finetune` is `True`, BatchNormalization runs in fine-tuning mode; it accumulates the input array to compute population statistics for normalization, and normalizes the input using batch statistics.

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)

Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.

- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

start_finetuning()
Resets the population count for collecting population statistics.

This method can be skipped if it is the first time to use the fine-tuning mode. Otherwise, this method should be called before starting the fine-tuning mode again.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

avg_mean = None

avg_var = None

beta = None

device
Device instance.

gamma = None

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *`chainerx`*.

chainer.links.BatchRenormalization

class chainer.links.**BatchRenormalization**(*size*, *rmax=1*, *dmax=0*, *decay=0.9*, *eps=2e-05*, *dtype=None*, *use_gamma=True*, *use_beta=True*, *initial_gamma=None*, *initial_beta=None*, *initial_avg_mean=None*, *initial_avg_var=None*)

Batch renormalization layer on outputs of linear or convolution functions.

This link wraps the *`batch_renormalization()`* and *`fixed_batch_renormalization()`* functions.

This is an extension of batch normalization, which ensures that the training and inference models generate the same outputs that depend on individual examples rather than the entire minibatch.

See: Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models

See also:

`batch_renormalization()`, *`fixed_batch_renormalization()`* *`batch_normalization()`*,

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*`str`*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*`int or tuple of ints`*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters link (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters mode (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

forward(*self, x, finetune=False*)

Invokes the forward propagation of BatchNormalization.

In training mode, the BatchNormalization computes moving averages of mean and variance for evaluation during training, and normalizes the input using batch statistics.

Parameters

- **x** (Variable) – Input variable.
- **finetune** (*bool*) – If it is in the training mode and finetune is True, BatchNormalization runs in fine-tuning mode; it accumulates the input array to compute population statistics for normalization, and normalizes the input using batch statistics.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

start_finetuning()

Resets the population count for collecting population statistics.

This method can be skipped if it is the first time to use the fine-tuning mode. Otherwise, this method should be called before starting the fine-tuning mode again.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

avg_mean = None

avg_var = None

beta = None

device
: `Device` instance.

gamma = None

local_link_hooks
: Ordered dictionary of registered link hooks.

  Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
: Generator of printable specs of this link.

  Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
: `True` if at least one parameter has an update rule enabled.

within_init_scope
: True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.

xp
: Array module corresponding to the device.

  Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.DecorrelatedBatchNormalization

class chainer.links.**DecorrelatedBatchNormalization**(*size*, *groups=16*, *decay=0.9*, *eps=2e-05*, *dtype=<class 'numpy.float32'>*)

Decorrelated batch normalization layer.

This link wraps the `decorrelated_batch_normalization()` and `fixed_decorrelated_batch_normalization()` functions. It works on outputs of linear or convolution functions.

It runs in three modes: training mode, fine-tuning mode, and testing mode.

In training mode, it normalizes the input by *batch statistics*. It also maintains approximated population statistics by moving averages, which can be used for instant evaluation in testing mode.

In fine-tuning mode, it accumulates the input to compute *population statistics*. In order to correctly compute the population statistics, a user must use this mode to feed mini-batches running through whole training dataset.

In testing mode, it uses pre-computed population statistics to normalize the input variable. The population statistics is approximated if it is computed by training mode, or accurate if it is correctly computed by fine-tuning mode.

Parameters

- **size** (*int or tuple of ints*) – Size (or shape) of channel dimensions.
- **groups** (*int*) – Number of groups to use for group whitening.
- **decay** (*float*) – Decay rate of moving average which is used during training.

- **eps** (*float*) – Epsilon value for numerical stability.
- **dtype** (*numpy.dtype*) – Type to use in computing.

See: Decorrelated Batch Normalization

See also:

decorrelated_batch_normalization(), *fixed_decorrelated_batch_normalization()*

Variables

- **avg_mean** (*N-dimensional array*) – Population mean.
- **avg_projection** (*N-dimensional array*) – Population projection.
- **groups** (*int*) – Number of groups to use for group whitening.
- **N** (*int*) – Count of batches given for fine-tuning.
- **decay** (*float*) – Decay rate of moving average which is used during training.
- **eps** (*float*) – Epsilon value for numerical stability. This value is added to the batch variances.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*self*, *x*, ***, *finetune=False*)

Invokes the forward propagation of DecorrelatedBatchNormalization.

In training mode, the DecorrelatedBatchNormalization computes moving averages of the mean and projection for evaluation during training, and normalizes the input using batch statistics.

Parameters

- **x** (*Variable*) – Input variable.
- **finetune** (*bool*) – If it is in the training mode and `finetune` is `True`, DecorrelatedBatchNormalization runs in fine-tuning mode; it accumulates the input array to compute population statistics for normalization, and normalizes the input using batch statistics.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

start_finetuning()

Resets the population count for collecting population statistics.

This method can be skipped if it is the first time to use the fine-tuning mode. Otherwise, this method should be called before starting the fine-tuning mode again.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *`init_scope()`* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *`chainerx`*.

chainer.links.GroupNormalization

class `chainer.links.`**`GroupNormalization`**(*groups*, *size=None*, *eps=1e-05*, *initial_gamma=None*, *initial_beta=None*)

Group normalization layer on outputs of convolution functions.

This link implements a "group normalization" which divides the channels into groups and computes within each group the mean and variance, then normalize by these statistics, scales and shifts them. Parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

Parameters

- **groups** (*int*) – The number of channel groups. This value must be a divisor of the number of channels.
- **size** (*int*) – Size of input units. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **eps** (*float*) – Epsilon value for numerical stability of normalization.
- **initial_gamma** (`Initializer`) – Initializer for scaling parameter. If `None`, then the vector is filled by 1. If a scalar, the vector is filled by it. If `numpy.ndarray`, the vector is set by it.
- **initial_beta** (`Initializer`) – Initializer for shifting parameter. If `None`, then the vector is filled by 0. If a scalar, the vector is filled by it. If `numpy.ndarray`, the vector is set by it.

Variables

- **groups** (*int*) – The number of channel groups.
- **gamma** (`Parameter`) – Scaling parameter.
- **beta** (`Parameter`) – Shifting parameter.
- **eps** (*float*) – Epsilon value for numerical stability.

See: Group Normalization

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Apply group normalization to given input.

Parameters **x** (`Variable`) – Batch tensors. First dimension of this value must be the size of minibatch and second dimension must be the number of channels. Moreover, this value must have one or more following dimensions, such as height and width.

Returns Output of the group normalization.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial

parameters but can be changed independently. `share` means all the elements which consist the resulting *`Sequential`* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

`serialize`(*serializer*)
Serializes the link object.

Parameters **`serializer`** (`AbstractSerializer`) – Serializer object.

`to_chx`()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

`to_cpu`()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

`to_device`(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **`device`** – Target device specifier. See *`get_device()`* for available values.

Returns: self

`to_gpu`(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **`device`** – Target device specifier. If omitted, the current device is used.

Returns: self

`to_intel64`()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

`zerograds`()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *`cleargrads()`* instead.

`__eq__``()`
Return self==value.

`__ne__``()`
Return self!=value.

`__lt__``()`
Return self<value.

`__le__``()`
Return self<=value.

`__gt__``()`
Return self>value.

`__ge__``()`
Return self>=value.

Attributes

`device`
`Device` instance.

`local_link_hooks`
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

`printable_specs`
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

`update_enabled`
`True` if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

`xp`
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.LayerNormalization

class `chainer.links.`**`LayerNormalization`**(*size=None*, *eps=1e-06*, *initial_gamma=None*, *initial_beta=None*)
Layer normalization layer on outputs of linear functions.

> **Warning:** This feature is experimental. The interface can change in the future.

This link implements a "layer normalization" layer which normalizes the input units by statistics that are computed along the second axis, scales and shifts them. Parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

Parameters

- **size** (*int*) – Size of input units. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **eps** (*float*) – Epsilon value for numerical stability of normalization.
- **initial_gamma** (`Initializer`) – Initializer for scaling vector. If `None`, then the vector is filled by 1. If a scalar, the vector is filled by it. If `numpy.ndarray`, the vector is set by it.
- **initial_beta** (`Initializer`) – Initializer for shifting vector. If `None`, then the vector is filled by 0. If a scalar, the vector is filled by it. If `numpy.ndarray`, the vector is set by it.

Variables

- **gamma** (`Parameter`) – Scaling parameter.
- **beta** (`Parameter`) – Shifting parameter.
- **eps** (*float*) – Epsilon value for numerical stability.

See: Layer Normalization

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

forward(*x*)

Apply layer normalization to given input.

Parameters **x** (Variable) – Batch vectors. Shape of this value must be *(batch_size, unit_size)*, e.g., the output of *linear()*.

Returns Output of the layer normalization.

Return type *Variable*

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

Device instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.BinaryHierarchicalSoftmax

class chainer.links.**BinaryHierarchicalSoftmax**(*in_size*, *tree*, *dtype=None*)
Hierarchical softmax layer over binary tree.

In natural language applications, vocabulary size is too large to use softmax loss. Instead, the hierarchical softmax uses product of sigmoid functions. It costs only $O(\log(n))$ time where n is the vocabulary size in average.

At first a user needs to prepare a binary tree whose each leaf is corresponding to a word in a vocabulary. When a word x is given, exactly one path from the root of the tree to the leaf of the word exists. Let $\mathrm{path}(x) = ((e_1, b_1), \ldots, (e_m, b_m))$ be the path of x, where e_i is an index of i-th internal node, and $b_i \in \{-1, 1\}$ indicates direction to move at i-th internal node (-1 is left, and 1 is right). Then, the probability of x is given as below:

$$
\begin{aligned}
P(x) &= \prod_{(e_i, b_i) \in \mathrm{path}(x)} P(b_i | e_i) \\
&= \prod_{(e_i, b_i) \in \mathrm{path}(x)} \sigma(b_i x^\top w_{e_i}),
\end{aligned}
$$

where $\sigma(\cdot)$ is a sigmoid function, and w is a weight matrix.

This function costs $O(\log(n))$ time as an average length of paths is $O(\log(n))$, and $O(n)$ memory as the number of internal nodes equals $n - 1$.

Parameters

- **in_size** (*int*) – Dimension of input vectors.
- **tree** – A binary tree made with tuples like *((1, 2), 3)*.
- **dtype** (*numpy.dtype*) – Type to use in computing.

Variables **W** (`Variable`) – Weight parameter matrix.

See: Hierarchical Probabilistic Neural Network Language Model [Morin+, AISTAT2005].

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)

Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

static create_huffman_tree (*word_counts*)

Makes a Huffman tree from a dictionary containing word counts.

This method creates a binary Huffman tree, that is required for *BinaryHierarchicalSoftmax*. For example, `{0: 8, 1: 5, 2: 6, 3: 4}` is converted to `((3, 1), (2, 0))`.

Parameters **word_counts** (*dict of int key and int or float values*) – Dictionary representing counts of words.

Returns Binary Huffman tree with tuples and keys of `word_coutns`.

delete_hook (*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*, *t*)
Computes the loss value for given input and ground truth labels.

Parameters

- **x** (`Variable`) – Input to the classifier at each node.
- **t** (`Variable`) – Batch of ground truth labels.

Returns Loss value.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all

elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (AbstractSerializer) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Attributes

`device`
`Device` instance.

`local_link_hooks`
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

`printable_specs`
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

`update_enabled`
`True` if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

`xp`
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.BlackOut

class `chainer.links.`**`BlackOut`**(*in_size*, *counts*, *sample_size*)
BlackOut loss layer.

See also:

`black_out()` for more detail.

Parameters

- **in_size** (*int*) – Dimension of input vectors.
- **counts** (*int list*) – Number of each identifiers.
- **sample_size** (*int*) – Number of negative samples.

Variables **W** (`Parameter`) – Weight parameter matrix.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

`addgrads`(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

`children`()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*, *t*)
Computes the loss value for given input and ground truth labels.

Parameters

- **x** (Variable) – Input of the weight matrix multiplication.
- **t** (Variable) – Batch of ground truth labels.

Returns Loss value.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.

- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

sample_data = None

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.CRF1d

class chainer.links.**CRF1d**(*n_label*, *initial_cost=None*)
Linear-chain conditional random field loss layer.

This link wraps the *crf1d()* function. It holds a transition cost matrix as a parameter.

Parameters

- **n_label** (*int*) – Number of labels.
- **initial_cost** (*initializer*) – Initializer to initialize the transition cost matrix. If this attribute is not specified, the transition cost matrix is initialized with zeros.

See also:

`crf1d()` for more detail.

Variables **cost** (`Variable`) – Transition cost parameter.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

argmax(*xs*, *transpose=False*)

Computes a state that maximizes a joint probability.

Parameters

- **xs** (*list of Variable*) – Input vector for each label.
- **transpose** (*bool*) – If `True`, input/output sequences
- **be sorted in descending order of length.** (*will*) –

Returns A tuple of *Variable* representing each log-likelihood and a list representing the argmax path.

Return type tuple

See also:

See `crf1d_argmax()` for more detail.

children()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise,

it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*xs*, *ys*, *reduce='mean'*, *transpose=False*)

Computes negative log-likelihood of linear-chain CRF

Parameters

- **xs** (*list of Variable*) – Input vector for each label
- **ys** (*list of Variable*) – Expected output labels.
- **transpose** (*bool*) – If `True`, input/output sequences
- **be sorted in descending order of length.** (*will*) –

Returns A variable holding the average negative log-likelihood of the input sequences.

Return type *Variable*

See also:

See `crf1d()` for more detail.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.SimplifiedDropconnect

class `chainer.links.`**SimplifiedDropconnect**(*in_size*, *out_size*, *ratio=0.5*, *nobias=False*, *initialW=None*, *initial_bias=None*)
Fully-connected layer with simplified dropconnect regularization.

Notice: This implementation cannot be used for reproduction of the paper. There is a difference between the current implementation and the original one. The original version uses sampling with gaussian distribution before passing activation function, whereas the current implementation averages before activation.

Parameters

- **in_size** (`int`) – Dimension of input vectors. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (`int`) – Dimension of output vectors.
- **nobias** (`bool`) – If `True`, then this link does not use the bias term.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 3.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias will be initialized to zero. When it is `numpy.ndarray`, its `ndim` should be 2.

Variables

- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

See also:

`simplified_dropconnect()`

See also:

Li, W., Matthew Z., Sixin Z., Yann L., Rob F. (2013). Regularization of Neural Network using DropConnect. International Conference on Machine Learning. URL

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*, *train=True*, *mask=None*, *use_batchwise_mask=True*)

Applies the simplified dropconnect layer.

Parameters

- **x** (chainer.Variable or *N-dimensional array*) – Batch of input vectors. Its first dimension `n` is assumed to be the *minibatch dimension*.
- **train** (*bool*) – If `True`, executes simplified dropconnect. Otherwise, simplified dropconnect link works as a linear unit.
- **mask** (None or chainer.Variable or *N-dimensional array*) – If `None`, randomized simplified dropconnect mask is generated. Otherwise, The mask must be `(n, M, N)` or `(M, N)` shaped array, and *use_batchwise_mask* is ignored. Main purpose of this option is debugging. *mask* array will be used as a dropconnect mask.
- **use_batchwise_mask** (*bool*) – If `True`, dropped connections depend on each sample in mini-batch.

Returns Output of the simplified dropconnect layer.

Return type *Variable*

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same

values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to chainer.thread_local.link_hooks, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the __init__(). This pair of key and value is used for representing this class or subclass with __str__().

update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns numpy, cupy or *chainerx*.

chainer.links.PReLU

class chainer.links.**PReLU**(*shape=()*, *init=0.25*)

Parametric ReLU function as a link.

Parameters

- **shape** (*tuple of ints*) – Shape of the parameter array.
- **init** (*float*) – Initial parameter value.

See the paper for details: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

To try PReLU instead of ReLU, replace `F.relu` with individual `PReLU` links registered to the model. For example, the model defined in the MNIST example can be rewritten as follows.

ReLU version (original):

```
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_units)
            self.l2 = L.Linear(None, n_units)
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

PReLU version:

```
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_units)
            self.a1 = L.PReLU()
            self.l2 = L.Linear(None, n_units)
            self.a2 = L.PReLU()
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = self.a1(self.l1(x))
        h2 = self.a2(self.l2(h1))
        return self.l3(h2)
```

See also:

chainer.functions.prelu()

Variables W (`Parameter`) – Coefficient of parametric ReLU.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies the parametric ReLU activation function.

Parameters **x** (`Variable`) – Input variable.

Returns Output of the parametric ReLU function.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *`get_device()`* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *`to_device()`* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *`cleargrads()`* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.Swish

class chainer.links.**Swish**(*beta_shape*, *beta_init=1.0*)
Swish activation function as a link.

Parameters

- **beta_shape** (*tuple of ints or None*) – Shape of the parameter variable β. If `None`, parameter initialization will be deferred until the first forward data pass at which time the shape will be determined.
- **beta_init** (*float*) – Initial value of the parameter variable β.

See the paper for details: Searching for Activation Functions

To try Swish instead of ReLU, replace `F.relu` with individual `Swish` links registered to the model. For example, the model defined in the MNIST example can be rewritten as follows.

ReLU version (original):

```
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_units)
            self.l2 = L.Linear(None, n_units)
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

Swish version:

```
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_units)
            self.s1 = L.Swish(None)
            self.l2 = L.Linear(None, n_units)
            self.s2 = L.Swish(None)
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = self.s1(self.l1(x))
        h2 = self.s2(self.l2(h1))
        return self.l3(h2)
```

See also:

See `chainer.functions.swish()` for the definition of Swish activation function.

Variables **beta** (`Parameter`) – Parameter variable β.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*x*)
Applies the Swish activation function.

Parameters **x** (`Variable`) – Input variable.

Returns Output of the Swish activation function.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.Maxout

class `chainer.links.`**Maxout**(*in_size*, *out_size*, *pool_size*, *initialW=None*, *initial_bias=0*)
Fully-connected maxout layer.

Let `M`, `P` and `N` be an input dimension, a pool size, and an output dimension, respectively. For an input vector x of size `M`, it computes

$$Y_i = \mathrm{max}_j (W_{ij\cdot} x + b_{ij}).$$

Here W is a weight tensor of shape `(M, P, N)`, b an optional bias vector of shape `(M, P)` and $W_{ij\cdot}$ is a sub-vector extracted from W by fixing first and second dimensions to i and j, respectively. Minibatch dimension is omitted in the above equation.

As for the actual implementation, this chain has a Linear link with a `(M * P, N)` weight matrix and an optional `M * P` dimensional bias vector.

Parameters

- **in_size** (`int`) – Dimension of input vectors.
- **out_size** (`int`) – Dimension of output vectors.
- **pool_size** (`int`) – Number of channels.
- **initialW** (*initializer*) – Initializer to initialize the weight. When it is `numpy.ndarray`, its `ndim` should be 3.
- **initial_bias** (*initializer*) – Initializer to initialize the bias. If `None`, the bias is omitted. When it is `numpy.ndarray`, its `ndim` should be 2.

Variables linear (`Link`) – The Linear link that performs affine transformation.

See also:

`maxout()`

See also:

Goodfellow, I., Warde-farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout Networks. In Proceedings of the 30th International Conference on Machine Learning (ICML-13) (pp. 1319-1327). URL

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **`hook`** (`LinkHook`) – Link hook to be registered.
- **`name`** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **`name`** (*`str`*) – Name of the child link. This name is also used as the attribute name.
- **`link`** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **`name`** (*`str`*) – Name of the parameter. This name is also used as the attribute name.
- **`shape`** (*`int or tuple of ints`*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **`dtype`** – Data type of the parameter array.
- **`initializer`** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

`add_persistent`(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **`name`** (*`str`*) – Name of the persistent value. This name is also used for the attribute name.
- **`value`** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*`str`*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *`initialize()`* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *`BatchNormalization`*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*`bool`*) – If `True`, persistent values are also copied. `True` by default.

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *`Parameter`*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward (*x*)
Applies the maxout layer.

Parameters **x** (Variable) – Batch of input vectors.

Returns Output of the maxout layer.

Return type *Variable*

from_chx ()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope ()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to

others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.NegativeSampling

class chainer.links.**NegativeSampling**(*in_size*, *counts*, *sample_size*, *power=0.75*, *dtype=None*)

Negative sampling loss layer.

This link wraps the `negative_sampling()` function. It holds the weight matrix as a parameter. It also builds a sampler internally given a list of word counts.

Parameters

- **in_size** (*int*) – Dimension of input vectors.
- **counts** (*int list*) – Number of each identifiers.
- **sample_size** (*int*) – Number of negative samples.
- **power** (*float*) – Power factor α.
- **dtype** (*numpy.dtype*) – Type to use in computing.

See also:

`negative_sampling()` for more detail.

Variables **W** (`Variable`) – Weight parameter matrix.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

forward(*x*, *t*, *reduce='sum'*, ***, *return_samples=False*)

Computes the loss value for given input and ground truth labels.

Parameters

- **x** (Variable) – Input of the weight matrix multiplication.
- **t** (Variable) – Batch of ground truth labels.
- **reduce** (*str*) – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, ValueError is raised.
- **return_samples** (*bool*) – If True, the sample array is also returned. The sample array is a (

Returns

If return_samples is False (default), loss value is returned.

Otherwise, a tuple of the loss value and the sample array is returned.

Return type *Variable* or tuple

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.3.3 Machine learning models

`chainer.links.Classifier`	A simple classifier model.

chainer.links.Classifier

class chainer.links.**Classifier**(*predictor*, *lossfun=<function softmax_cross_entropy>*, *accfun=<function accuracy>*, *label_key=-1*)
A simple classifier model.

This is an example of chain that wraps another chain. It computes the loss and accuracy based on a given input/label pair.

Parameters

- **predictor** (`Link`) – Predictor network.
- **lossfun** (*callable*) – Loss function. You can specify one of loss functions from *built-in loss functions*, or your own loss function (see the example below). It should not be an *loss functions with parameters* (i.e., `Link` instance). The function must accept two argument (an output from predictor and its ground truth labels), and return a loss. Returned value must be a Variable derived from the input Variable to perform backpropagation on the variable.
- **accfun** (*callable*) – Function that computes accuracy. You can specify one of evaluation functions from *built-in evaluation functions*, or your own evaluation function. The signature of the function is the same as `lossfun`.
- **label_key** (*int or str*) – Key to specify label variable from arguments. When it is `int`, a variable in positional arguments is used. And when it is `str`, a variable in keyword arguments is used.

Variables

- **predictor** (`Link`) – Predictor network.

- **lossfun** (*callable*) – Loss function. See the description in the arguments for details.
- **accfun** (*callable*) – Function that computes accuracy. See the description in the arguments for details.
- **y** (Variable) – Prediction for the last minibatch.
- **loss** (Variable) – Loss value for the last minibatch.
- **accuracy** (Variable) – Accuracy for the last minibatch.
- ***compute_accuracy*** (*bool*) – If True, compute accuracy on the forward computation. The default value is True.

Note: This link uses `chainer.softmax_cross_entropy()` with default arguments as a loss function (specified by `lossfun`), if users do not explicitly change it. In particular, the loss function does not support double backpropagation. If you need second or higher order differentiation, you need to turn it on with `enable_double_backprop=True`:

```
>>> import chainer.functions as F
>>> import chainer.links as L
>>>
>>> def lossfun(x, t):
...     return F.softmax_cross_entropy(
...         x, t, enable_double_backprop=True)
>>>
>>> predictor = L.Linear(10)
>>> model = L.Classifier(predictor, lossfun=lossfun)
```

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters name (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**args*, ***kwargs*)
Computes the loss value for an input and label pair.

It also computes accuracy and stores it to the attribute.

Parameters

- **args** (*list of ~chainer.Variable*) – Input minibatch.
- **kwargs** (*dict of ~chainer.Variable*) – Input minibatch.

When `label_key` is `int`, the corresponding element in `args` is treated as ground truth labels. And when it is `str`, the element in `kwargs` is used. The all elements of `args` and `kwargs` except the ground truth labels are features. It feeds features to the predictor and compare the result with ground truth labels.

Note: We set `None` to the attributes `y`, `loss` and `accuracy` each time before running the predictor, to avoid unnecessary memory consumption. Note that the variables set on those attributes hold the whole computation graph when they are computed. The graph stores interim values on memory required for back-propagation. We need to clear the attributes to free those values.

Returns Loss value.

Return type *Variable*

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

compute_accuracy = True

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.3.4 Pre-trained models

Pre-trained models are mainly used to achieve a good performance with a small dataset, or extract a semantic feature vector. Although `CaffeFunction` automatically loads a pre-trained model released as a caffemodel, the following link models provide an interface for automatically converting caffemodels, and easily extracting semantic feature vectors.

For example, to extract the feature vectors with `VGG16Layers`, which is a common pre-trained model in the field of image recognition, users need to write the following few lines:

```
from chainer.links import VGG16Layers
from PIL import Image

model = VGG16Layers()
img = Image.open("path/to/image.jpg")
feature = model.extract([img], layers=["fc7"])["fc7"]
```

where `fc7` denotes a layer before the last fully-connected layer. Unlike the usual links, these classes automatically load all the parameters from the pre-trained models during initialization.

VGG Networks

chainer.links.VGG16Layers	A pre-trained CNN model with 16 layers provided by VGG team.
chainer.links.VGG19Layers	A pre-trained CNN model with 19 layers provided by VGG team.
chainer.links.model.vision.vgg.prepare	Converts the given image to the numpy array for VGG models.

chainer.links.VGG16Layers

class chainer.links.**VGG16Layers**(*pretrained_model='auto'*)

A pre-trained CNN model with 16 layers provided by VGG team.

During initialization, this chain model automatically downloads the pre-trained caffemodel, convert to another chainer model, stores it on your local directory, and initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector from a given image, or fine-tune the model on a different dataset. Note that this pre-trained model is released under Creative Commons Attribution License.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

See: K. Simonyan and A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition

Parameters **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically downloads the caffemodel from the internet. Note that in this case the converted chainer model is stored on `$CHAINER_DATASET_ROOT/pfnet/chainer/models` directory, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value as a environment variable. The converted chainer model is automatically used from the second time. If the argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.Normal(scale=0.01)`.

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args, **kwargs*)

Call self as a function.

__getitem__(*name*)

Equivalent to getattr.

add_hook(*hook, name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.

- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz(*path_caffemodel*, *path_npz*)
Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract(*self, images, layers=['fc7'], size=(224, 224)*)

Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of VGGLayers (16 or 19 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)

Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract. If `None`, 'prob' will be used as layers.

Returns A dictionary in which the key contains the layer and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict (*images, oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.
- **oversample** (*bool*) – If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.VGG19Layers

class chainer.links.**VGG19Layers**(*pretrained_model='auto'*)
A pre-trained CNN model with 19 layers provided by VGG team.

During initialization, this chain model automatically downloads the pre-trained caffemodel, convert to another chainer model, stores it on your local directory, and initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector from a given image, or fine-tune the model on a different dataset. Note that this pre-trained model is released under Creative Commons Attribution License.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

See: K. Simonyan and A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition

Parameters **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically downloads the caffemodel from the internet. Note that in this case the converted chainer model is stored on `$CHAINER_DATASET_ROOT/pfnet/chainer/models` directory, where

`$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value as a environment variable. The converted chainer model is automatically used from the second time. If the argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.Normal(scale=0.01)`.

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz(*path_caffemodel*, *path_npz*)
Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract(*self, images, layers=['fc7'], size=(224, 224)*)

Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of VGGLayers (16 or 19 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)
Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract. If `None`, 'prob' will be used as layers.

Returns A dictionary in which the key contains the layer and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)

Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict (*images*, *oversample=True*)

Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.
- **oversample** (*bool*) – If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same

values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns numpy, cupy or *chainerx*.

chainer.links.model.vision.vgg.prepare

chainer.links.model.vision.vgg.**prepare**(*image*, *size=(224, 224)*)

Converts the given image to the numpy array for VGG models.

Note that you have to call this method before forward because the pre-trained vgg model requires to resize the given image, covert the RGB to the BGR, subtract the mean, and permute the dimensions before calling.

Parameters

- **image** (*PIL.Image or numpy.ndarray*) – Input image. If an input is numpy.ndarray, its shape must be (height, width), (height, width, channels), or (channels, height, width), and the order of the channels must be RGB.
- **size** (*pair of ints*) – Size of converted images. If None, the given image is not resized.

Returns The converted output array.

Return type numpy.ndarray

Note: ChainerCV contains implementation of VGG networks as well (i.e., chainercv.links.model.vgg.VGG16). Unlike the Chainer's implementation, the ChainerCV's implementation assumes the color channel of the input image to be ordered in RGB instead of BGR.

GoogLeNet

chainer.links.GoogLeNet	A pre-trained GoogLeNet model provided by BVLC.
chainer.links.model.vision.googlenet.prepare	Converts the given image to the numpy array for GoogLeNet.

chainer.links.GoogLeNet

class chainer.links.**GoogLeNet**(*pretrained_model='auto'*)

A pre-trained GoogLeNet model provided by BVLC.

When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use convert_caffemodel_to_npz classmethod instead.

GoogLeNet, which is also called Inception-v1, is an architecture of convolutional neural network proposed in 2014. This model is relatively lightweight and requires small memory footprint during training compared with modern architectures such as ResNet. Therefore, if you fine-tune your network based on a model pre-trained by Imagenet and need to train it with large batch size, GoogLeNet may be useful. On the other hand, if you just want an off-the-shelf classifier, we recommend that you use ResNet50 or other models since they are more accurate than GoogLeNet.

The original model is provided here: https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

Parameters **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically downloads the caffemodel from the internet. Note that in this case the converted chainer model is stored on `$CHAINER_DATASET_ROOT/pfnet/chainer/models` directory, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value as a environment variable. The converted chainer model is automatically used from the second time. If the argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in BVLC, i.e., `chainer.initializers.LeCunUniform(scale=1.0)`. Note that, in Caffe, when weight_filler is specified as "xavier" type without variance_norm parameter, the weights are initialized by Uniform(-s, s), where $s = \sqrt{\frac{3}{fan_{in}}}$ and fan_{in} is the number of input units. This corresponds to LeCunUniform in Chainer but not GlorotUniform.

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a

scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children ()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz (*path_caffemodel*, *path_npz*)

Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy (*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract (*self, images, layers=['pool5'], size=(224, 224)*)
Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of `GoogLeNet`
with chainer.using_config('train', False):
```

(continues on next page)

(continued from previous page)

```
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)

Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict (*images, oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.
- **oversample** (*bool*) – If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```python
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

available_layers

device

`Device` instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.model.vision.googlenet.prepare

`chainer.links.model.vision.googlenet.`**`prepare`**`(`*image*, *size=(224, 224)*`)`
Converts the given image to the numpy array for GoogLeNet.

Note that you have to call this method before `forward` because the pre-trained GoogLeNet model requires to resize the given image, covert the RGB to the BGR, subtract the mean, and permute the dimensions before calling.

Parameters

- **image** (*PIL.Image or numpy.ndarray*) – Input image. If an input is `numpy.ndarray`, its shape must be `(height, width)`, `(height, width, channels)`, or `(channels, height, width)`, and the order of the channels must be RGB.
- **size** (*pair of ints*) – Size of converted images. If `None`, the given image is not resized.

Returns The converted output array.

Return type numpy.ndarray

Residual Networks

`chainer.links.model.vision.resnet.ResNetLayers`	A pre-trained CNN model provided by MSRA.
`chainer.links.ResNet50Layers`	A pre-trained CNN model with 50 layers provided by MSRA.
`chainer.links.ResNet101Layers`	A pre-trained CNN model with 101 layers provided by MSRA.

Continued on next page

Table 21 – continued from previous page

chainer.links.ResNet152Layers	A pre-trained CNN model with 152 layers provided by MSRA.
chainer.links.model.vision.resnet.prepare	Converts the given image to a numpy array for ResNet.

chainer.links.model.vision.resnet.ResNetLayers

class chainer.links.model.vision.resnet.**ResNetLayers**(*pretrained_model*, *n_layers*, *downsample_fb=False*)

A pre-trained CNN model provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a `.npz` file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike `VGG16Layers`, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

- **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically loads and converts the caffemodel from `$CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-{n-layers}-model.caffemodel`, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value by modifying the environment variable and {n_layers} is replaced with the specified number of layers given as the first argument to this constructor. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.HeNormal(scale=1.0)`.
- **n_layers** (*int*) – The number of layers of this model. It should be either 50, 101, or 152.
- **downsample_fb** (*bool*) – If this argument is specified as `False`, it performs downsampling by placing stride 2 on the 1x1 convolutional layers (the original MSRA ResNet). If this argument is specified as `True`, it performs downsampling by placing stride 2 on the 3x3 convolutional layers (Facebook ResNet).

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If None, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz (*path_caffemodel*, *path_npz*, *n_layers=50*)
Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params ()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract (*self, images, layers=['pool5'], size=(224, 224)*)
Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of ResNetLayers (50 or 101 or 152 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward (*self, x, layers=['prob']*)
Computes all the feature maps specified by `layers`.

Parameters

- **x** (Variable) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict(*images, oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.

- **oversample** (*bool*) – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)
Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.ResNet50Layers

class `chainer.links.`**ResNet50Layers**(*pretrained_model='auto'*, *downsample_fb=False*)
A pre-trained CNN model with 50 layers provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a `.npz` file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike `VGG16Layers`, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

ResNet50 has 25,557,096 trainable parameters, and it's 58% and 43% fewer than ResNet101 and ResNet152, respectively. On the other hand, the top-5 classification accuracy on ImageNet dataset drops only 0.7% and 1.1% from ResNet101 and ResNet152, respectively. Therefore, ResNet50 may have the best balance between the accuracy and the model size. It would be basically just enough for many cases, but some advanced models for object detection or semantic segmentation use deeper ones as their building blocks, so these deeper ResNets are here for making reproduction work easier.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

- **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically loads and converts the caffemodel from `$CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-50-model.caffemodel`, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.HeNormal(scale=1.0)`.
- **downsample_fb** (*bool*) – If this argument is specified as `False`, it performs downsampling by placing stride 2 on the 1x1 convolutional layers (the original MSRA ResNet). If this argument is specified as `True`, it performs downsampling by placing stride 2 on the 3x3 convolutional layers (Facebook ResNet).

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

`add_link`(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

`add_param`(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()

Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz(*path_caffemodel*, *path_npz*, *n_layers=50*)

Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy(*mode='share'*)

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link.

copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default mode is share.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using copy.deepcopy(). The old behavior (not copying persistent values) can be reproduced with copy_persistent=False.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If True, persistent values are also copied. True by default.

count_params ()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook (*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept (*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to False.

enable_update ()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

extract (*self, images, layers=['pool5'], size=(224, 224)*)

Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of ResNetLayers (50 or 101 or 152 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)

Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict (*images, oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a numpy.ndarray, make sure that color order is RGB.
- **oversample** (*bool*) – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.ResNet101Layers

class `chainer.links.`**ResNet101Layers**(*pretrained_model='auto'*, *downsample_fb=False*)
A pre-trained CNN model with 101 layers provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a `.npz` file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike `VGG16Layers`, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

ResNet101 has 44,549,224 trainable parameters, and it's 43% fewer than ResNet152 model, while the top-5 classification accuracy on ImageNet dataset drops 1.1% from ResNet152. For many cases, ResNet50 may have the best balance between the accuracy and the model size.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

- **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically loads and converts the caffemodel from `$CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-101-model.caffemodel`, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.HeNormal(scale=1.0)`.

- **downsample_fb** (*bool*) – If this argument is specified as `False`, it performs downsampling by placing stride 2 on the 1x1 convolutional layers (the original MSRA ResNet). If this argument is specified as `True`, it performs downsampling by placing stride 2 on the 3x3 convolutional layers (Facebook ResNet).

Variables **available_layers** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz (*path_caffemodel*, *path_npz*, *n_layers=50*)
Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.

- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract(*self, images, layers=['pool5'], size=(224, 224)*)

Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of ResNetLayers (50 or 101 or 152 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)

Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)

Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict (*images*, *oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.
- **oversample** (*bool*) – If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.ResNet152Layers

class chainer.links.**ResNet152Layers**(*pretrained_model='auto'*, *downsample_fb=False*)

A pre-trained CNN model with 152 layers provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a `.npz` file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike `VGG16Layers`, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use `convert_caffemodel_to_npz` classmethod instead.

ResNet152 has 60,192,872 trainable parameters, and it's the deepest ResNet model and it achieves the best result on ImageNet classification task in ILSVRC 2015.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

- **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a `.npz` file. If this argument is specified as `auto`, it automatically loads and converts the caffemodel from `$CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-152-model.caffemodel`, where `$CHAINER_DATASET_ROOT` is set as `$HOME/.chainer/dataset` unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as `None`, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., `chainer.initializers.HeNormal(scale=1.0)`.
- **downsample_fb** (*bool*) – If this argument is specified as `False`, it performs downsampling by placing stride 2 on the 1x1 convolutional layers (the original MSRA ResNet). If this argument is specified as `True`, it performs downsampling by placing stride 2 on the 3x3 convolutional layers (Facebook ResNet).

Variables ***available_layers*** (*list of str*) – The list of available layer names used by `forward` and `extract` methods.

Methods

__call__(**args*, ***kwargs*)

Call self as a function.

__getitem__(*name*)

Equivalent to getattr.

add_hook(*hook*, *name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link (*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz (*path_caffemodel*, *path_npz*, *n_layers=50*)
Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extract(*self, images, layers=['pool5'], size=(224, 224)*)
Extracts all the feature maps of given images.

The difference of directly executing `forward` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `forward` functions.

Unlike `predict` method, this method does not override `chainer.config.train` and `chainer.config.enable_backprop` configuration. If you want to extract features without updating model parameters, you need to manually set configuration when calling this method as follows:

```
# model is an instance of ResNetLayers (50 or 101 or 152 layers)
with chainer.using_config('train', False):
    with chainer.using_config('enable_backprop', False):
        feature = model.extract([image])
```

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
- **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

forward(*self, x, layers=['prob']*)
Computes all the feature maps specified by `layers`.

Parameters

- **x** (`Variable`) – Input variable. It should be prepared by `prepare` function.
- **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

predict(*images, oversample=True*)
Computes all the probabilities of given images.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images. When you specify a color image as a `numpy.ndarray`, make sure that color order is RGB.
- **oversample** (*bool*) – If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type *Variable*

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()
Return self>=value.

Attributes

available_layers

device
Device instance.

functions

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.links.model.vision.resnet.prepare

`chainer.links.model.vision.resnet.`**prepare**(*image*, *size=(224, 224)*)
Converts the given image to a numpy array for ResNet.

Note that this method must be called before calling `forward`, because the pre-trained resnet model will resize the given image, convert from RGB to BGR, subtract the mean, and permute the dimensions before calling.

Parameters

- **image** (*PIL.Image or numpy.ndarray*) – Input image. If an input is `numpy.ndarray`, its shape must be `(height, width)`, `(height, width, channels)`, or `(channels, height, width)`, and the order of the channels must be RGB.
- **size** (*pair of ints*) – Size of converted images. If `None`, the given image is not resized.

Returns The converted output array.

Return type numpy.ndarray

Note: ChainerCV contains implementation of ResNet as well (i.e., `chainercv.links.model.resnet.ResNet50`, `chainercv.links.model.resnet.ResNet101`, `chainercv.links.model.resnet.ResNet152`). Unlike the Chainer's implementation, the ChainerCV's implementation assumes the color channel of the input image to be ordered in RGB instead of BGR.

ChainerCV models

Note: ChainerCV supports implementations of links that are useful for computer vision problems, such as object detection, semantic segmentation, and instance segmentation. The documentation can be found in `chainercv.links`. Here is a subset of models with pre-trained weights supported by ChainerCV:

- **Detection**
  - `chainercv.links.model.faster_rcnn.FasterRCNNVGG16`
  - `chainercv.links.model.ssd.SSD300`
  - `chainercv.links.model.ssd.SSD512`
  - `chainercv.links.model.yolo.YOLOv2`
  - `chainercv.links.model.yolo.YOLOv3`
- **Semantic Segmentation**
  - `chainercv.links.model.segnet.SegNetBasic`
  - `chainercv.experimental.links.model.pspnet.PSPNetResNet101`
- **Instance Segmentation**
  - `chainercv.experimental.links.model.fcis.FCISResNet101`
- **Classification**
  - `chainercv.links.model.resnet.ResNet101`
  - `chainercv.links.model.resnet.ResNet152`
  - `chainercv.links.model.resnet.ResNet50`
  - `chainercv.links.model.senet.SEResNet101`
  - `chainercv.links.model.senet.SEResNet152`
  - `chainercv.links.model.senet.SEResNet50`
  - `chainercv.links.model.senet.SEResNeXt101`
  - `chainercv.links.model.senet.SEResNeXt50`
  - `chainercv.links.model.vgg.VGG16`

Compatibility with other frameworks

chainer.links.TheanoFunction	Theano function wrapper.
chainer.links.caffe.CaffeFunction	Caffe emulator based on the model file of Caffe.

chainer.links.TheanoFunction

class chainer.links.**TheanoFunction**(*inputs*, *outputs*)

Theano function wrapper.

Warning: This feature is experimental. The interface can change in the future.

This function wraps Theano function as a *chainer.Link*. A user needs to make input Theano variables and output Theano variables. This function automatically creates Theano function for forward calculation and backward calculation from inputs and ouptuts. And then, it sends data in *chainer.Variable* to the function and gets results from Theano.

Example

```
>>> import theano
>>> x = theano.tensor.fvector()
>>> y = theano.tensor.fvector()
>>> z = x + y
>>> w = x - y
>>> f = L.TheanoFunction(inputs=[x, y], outputs=[z, w])
>>> a = chainer.Variable(np.array([1, 2], dtype=np.float32))
>>> b = chainer.Variable(np.array([2, 3], dtype=np.float32))
>>> c, d = f(a, b)
>>> c.array
array([3., 5.], dtype=float32)
>>> d.array
array([-1., -1.], dtype=float32)
```

Note: The current implementation always copies `cupy.ndarray` to CPU.

Parameters

- **inputs** (tuple of `theano.tensor.TensorVariable`) – Input variables of Theano. This function accepts the same number of *Variable*s in forward computation.
- **outputs** (tuple of `theano.tensor.TensorVariable`) – Output variables of Theano. The function returns the same number of *Variable*s as `outputs`.

Methods

__call__(**args*, ***kwargs*)

Call self as a function.

add_hook(*hook*, *name=None*)

Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays

are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(**args*)

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat(*n_repeat, mode='init'*)
Repeats this link multiple times to make a `Sequential`.

This method returns a `Sequential` object which has the same `Link` multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer `Sequential` block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize(*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks

Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs

Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled

`True` if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp

Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.links.caffe.CaffeFunction

class chainer.links.caffe.**CaffeFunction**(*model_path*)

Caffe emulator based on the model file of Caffe.

Given a protocol buffers file of a Caffe model, this class loads and emulates it on `Variable` objects. It supports the official reference models provided by BVLC.

Note: CaffeFunction ignores the following layers:

- Layers that CaffeFunction does not support (including data layers)
- Layers that have no top blobs
- Layers whose bottom blobs are incomplete (i.e., some or all of them are not given nor computed)

Warning: It does not support full compatibility against Caffe. Some layers and configurations are not implemented in Chainer yet, though the reference models provided by the BVLC team are supported except data layers.

Example

Consider we want to extract the (unnormalized) log class probability of given images using BVLC reference CaffeNet. The model can be downloaded from:

http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel

We want to compute the `fc8` blob from the `data` blob. It is simply written as follows:

```
# Load the model
func = CaffeFunction('path/to/bvlc_reference_caffenet.caffemodel')

# Minibatch of size 10
x_data = numpy.ndarray((10, 3, 227, 227), dtype=numpy.float32)
...  # (Fill the minibatch here)

# Forward the pre-trained net
x = Variable(x_data)
y, = func(inputs={'data': x}, outputs=['fc8'])
```

The result `y` contains the Variable corresponding to the `fc8` blob. The computational graph is memorized as a usual forward computation in Chainer, so we can run backprop through this pre-trained net.

Parameters **model_path** (*str*) – Path to the binary-proto model file of Caffe.

Variables **forwards** (*dict*) – A mapping from layer names to corresponding functions.

Methods

__call__(**args, **kwargs*)
Call self as a function.

__getitem__(*name*)
Equivalent to getattr.

add_hook(*hook, name=None*)
Registers a link hook.

Parameters

- **hook** (LinkHook) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*name, link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param(*name, shape=None, dtype=<class 'numpy.float32'>, initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.

- **initializer** (*initializer*) – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument mode below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either init, copy, or share. init means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. copy means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. share means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default mode is share.

Returns Copied link object.

Return type *Link*

copyparams (*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()

Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)

Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

forward(*self*, *inputs*, *outputs*, *disable=()*)

Executes a sub-network of the network.

This function acts as an interpreter of the network definition for Caffe. On execution, it interprets each layer one by one, and if the bottom blobs are already computed, then emulates the layer and stores output blobs as `Variable` objects.

Parameters

- **inputs** (*dict*) – A dictionary whose key-value pairs indicate initial correspondences between blob names and `Variable` objects.
- **outputs** (*Iterable*) – A list of blob names whose corresponding `Variable` objects are returned.
- **disable** (*Iterable*) – A list of layer names that will be ignored during the forward computation.

Returns A tuple of output `Variable` objects corresponding to elements of the *outputs* argument.

Return type tuple

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (`str`) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)
Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.3.5 Link and Chain base classes

`chainer.Link`	Building block of model definitions.
`chainer.Chain`	Composable link with object-like interface.
`chainer.ChainList`	Composable link with list-like interface.
`chainer.Sequential`	Sequential model which has a single-stream forward pass.

chainer.Link

class chainer.**Link**(***params*)
Building block of model definitions.

Link is a building block of neural network models that support various features like handling parameters, defining network fragments, serialization, etc.

Link is the primitive structure for the model definitions. It supports management of parameter variables and *persistent values* that should be incorporated to serialization.

Parameter is an instance of `Parameter` registered to a link. A `Parameter` object can be registered as a parameter of the link by assigning it to an attribute within *an initialization scope*, which is a code surrounded by a `init_scope()` context manager using the `with` statement.

Persistent values are arrays, scalars, or any other serializable values registered via `register_persistent()` or `add_persistent()`.

Note: Whereas arbitrary serializable objects can be registered as persistent values, it is strongly recommended that you just register values that should be treated as results of learning. A typical example of persistent values is ones computed during training and required for testing, e.g. running statistics for batch normalization.

Parameters and persistent values are referred by their names. They can be accessed as attributes of the links. Link class itself manages the lists of names of parameters and persistent values to distinguish parameters and persistent values from other attributes.

Link can be composed into more complex models. This composition feature is supported by child classes like `Chain` and `ChainList`. One can create a chain by combining one or more links. See the documents for these classes for details.

As noted above, Link supports the serialization protocol of the `Serializer` class. **Note that only parameters and persistent values are saved and loaded.** Other attributes are considered as a part of user program (i.e. a part of network definition). In order to construct a link from saved file, other attributes must be identically reconstructed by user codes.

Example

This is a simple example of custom link definition. Chainer itself also provides many links defined under the `links` module. They might serve as examples, too.

Consider we want to define a simple primitive link that implements a fully-connected layer based on the `linear()` function. Note that this function takes input units, a weight variable, and a bias variable as arguments. Then, the fully-connected layer can be defined as follows:

```
import chainer
import chainer.functions as F
from chainer import initializers
import numpy as np

class LinearLayer(chainer.Link):

    def __init__(self, n_in, n_out):
        super(LinearLayer, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(
                initializers.Normal(), (n_out, n_in))
            self.b = chainer.Parameter(
                initializers.Zero(), (n_out,))

    def forward(self, x):
        return F.linear(x, self.W, self.b)
```

This example shows that a user can define arbitrary parameters and use them in any methods. Links typically implement the `forward` operator, although they can also provide other methods to implement the forward propagation.

Parameters **params** – Names, shapes, and optional dtypes of initial parameters. The keywords are used as the parameter names and the corresponding values consist either of the shape or a tuple of shape and a dtype `(shape, dtype)`. If only the shape is supplied, the default dtype will be used.

Variables **name** (`str`) – Name of this link, given by the parent chain (if exists).

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their `initialize()` method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

repeat (*n_repeat, mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.Chain

class chainer.**Chain**(***links*)
Composable link with object-like interface.

Composability is one of the most important features of neural nets. Neural net models consist of many reusable fragments, and each model itself might be embedded into a larger learnable system. Chain enables us to write a neural net based on composition, without bothering about routine works like collecting parameters, serialization, copying the structure with parameters shared, etc.

This class actually provides a way to compose one or more links into one structure. A chain can contain one or more *child links*. Child link is a link registered to the chain with its own name. The child link is stored to an attribute of the chain with the name. User can write a whole model or a fragment of neural nets as a child class of Chain.

Each chain itself is also a link. Therefore, one can combine chains into higher-level chains. In this way, links and chains construct a *link hierarchy*. Link hierarchy forms a tree structure, where each node is identified by the path from the root. The path is represented by a string like a file path in UNIX, consisting of names of nodes on the path, joined by slashes `/`.

A child link can be added just by assigning it to an attribute of the chain within *init_scope()*.

The registered child link is saved and loaded on serialization and deserialization, and involved in the optimization. The registered link is called a child. The child link is accessible via `children()` generator, which returns a generator running through the children in lexical order.

On registration of a child link, its `name` attribute is also set (or overwritten if the link has already been registered to another chain).

Example

This is a simple example of custom chain definition. Chainer itself also provides some chains defined under the `links` module. They might serve as examples, too.

Consider we want to define a multi-layer perceptron consisting of two hidden layers with rectifiers as activation functions. We can use the `Linear` link as a building block:

```
import chainer
import chainer.functions as F
import chainer.links as L

class MultiLayerPerceptron(chainer.Chain):

    def __init__(self, n_in, n_hidden, n_out):
        super(MultiLayerPerceptron, self).__init__()
        with self.init_scope():
            self.layer1 = L.Linear(n_in, n_hidden)
            self.layer2 = L.Linear(n_hidden, n_hidden)
            self.layer3 = L.Linear(n_hidden, n_out)

    def forward(self, x):
        # Forward propagation
        h1 = F.relu(self.layer1(x))
        h2 = F.relu(self.layer2(h1))
        return self.layer3(h2)
```

Child links are registered via the assignment within a `with self.init_scope():` block. The forward propagation is often implemented as the `forward` operator as the above example, though it is not mandatory.

Parameters **`links`** – Child links. The keywords are used as their names. The names are also set to the links.

Methods

`__call__`(**args*, ***kwargs*)
Call self as a function.

`__getitem__`(*name*)
Equivalent to getattr.

`add_hook`(*hook*, *name=None*)
Registers a link hook.

Parameters

- **`hook`** (`LinkHook`) – Link hook to be registered.
- **`name`** (*`str`*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link (*name*, *link*)
Registers a child link to this chain.

Parameters

- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (Link) – Source link object.

children ()
Returns a generator of all child links.

Returns A generator object that generates all child links.

cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy (*mode='share'*)
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. There are three modes to perform copy. Please see the documentation for the argument `mode` below.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Parameters **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameter variables under the returned link object is re-initialized by calling their *initialize()* method, so that all the parameters may have different initial values from the original link. `copy` means that the link object is deeply copied, so that its parameters are not re-initialized but are also deeply copied. Thus, all parameters have same initial values but can be changed independently. `share` means that the link is shallowly copied, so that its parameters' arrays are shared with the original one. Thus, their values are changed synchronously. The default `mode` is `share`.

Returns Copied link object.

Return type *Link*

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (`str`) – Name of the attribute to be registered.

repeat (*n_repeat*, *mode='init'*)

Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. share means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

serialize (*serializer*)

Serializes the link object.

Parameters **serializer** (AbstractSerializer) – Serializer object.

to_chx ()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu ()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()

Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

chainer.ChainList

class chainer.**ChainList**(**links*)
Composable link with list-like interface.

This is another example of compositional link. Unlike `Chain`, this class can be used like a list of child links. Each child link is indexed by a non-negative integer, and it maintains the current number of registered child links. The `add_link()` method inserts a new link at the end of the list. It is useful to write a chain with arbitrary number of child links, e.g. an arbitrarily deep multi-layer perceptron.

This class inherits the methods *index*, *count*, *append*, *reverse*, *extend*, *pop*, *remove* from *collections.abc.MutableSequence* and can be accessed and assigned by index or slice.

Parameters **links** – Initial child links.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*index*)
Returns the child at given index.

Parameters **index** (`int`) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*index*, *value*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append(*value*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of *BatchNormalization*). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (Link) – Source link object.
- **copy_persistent** (*bool*) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the *Parameter*s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters name (*str*) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters visitor (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

init_scope()
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert(*index*, *link*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (`Link`) – The link to be inserted.

links(*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks(*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent(*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

repeat(*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The mode argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The net object contains 16 blocks, each of which is ConvBNReLU. And the mode was init, so each block is re-initialized with different parameters. If you give copy to this argument, each block has same values for its parameters but its object ID is different from others. If it is share, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either init, copy, or share. init means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. copy means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial

parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()
S.reverse() – reverse *IN PLACE*

serialize(*serializer*)
Serializes the link object.

Parameters serializer (`AbstractSerializer`) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient *cleargrads()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.Sequential

class chainer.**Sequential**(**layers*)
Sequential model which has a single-stream forward pass.

Warning: This feature is experimental. The interface can change in the future.

This class enables to construct a network which has sequential structure easily. While `Chain` and `ChainList` can only take `Link` object as input to their constructor, this `Sequential` can take arbitrary number of any callable objects for the forward pass computation. A `Sequential` calls the given callable objects sequentially inside of the `forward()` method in the same order as the given arguments. Therefore, you do not need to write the forward pass computation explicitly.

Example

The below example code shows how to use this class to construct a simple sequential network:

```
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import Sequential

# Model definition without writing forward function
model = Sequential(
    L.Linear(n_in, n_hidden),
    F.relu,
    L.Linear(n_hidden, n_hidden),
    F.relu,
    L.Linear(n_hidden, n_out)
)

# Compute the forward pass
y = model(x)
```

where `x` denotes a mini-batch of `n_in`-dimensional input vectors.

Furthermore, `Sequential` supports built-in list APIs, so you can concatenate `Sequential` objects to create a longer `Sequential` model easily with the same ways as Python lists:

```
>>> from chainer import Sequential
>>> model_A = Sequential(L.Linear(10, 10), F.relu)
>>> model_B = Sequential(L.Linear(10, 10), F.sigmoid)
>>> model_C = model_A + model_B
```

To repeat a `Sequential` object multiple times, you can use `repeat()` method.

```
>>> model_D = model_A.repeat(3)
```

You can also add your own functions or any callable objects to a `Sequential` object:

```
from chainer.links.model.vision.vgg import VGG16Layers

model = Sequential()
model.append(L.Linear(n_out, n_hidden))
model.append(F.relu)
model.append(lambda x: F.reshape(x, (1, 3, 224, 224)))
model.append(VGG16Layers())
model.append(lambda x: x['prob'])

y = model(x)
```

The above code example shows how to add some layers to the `model` using *append()* method and then add a large network (`VGG16Layers`) and finally add a lambda function to extract the `prob` output.

You can check the structure of your model briefly using `print` as following:

```
>>> print(model_C)
Sequential(
  (0): Linear(in_size=10, out_size=10, nobias=False),
  (1): <function relu at 0x...>,
  (2): Linear(in_size=10, out_size=10, nobias=False),
  (3): <function sigmoid at 0x...>,
)
```

Note: Note that a *Sequential* link which has at least one `lambda` function as its member cannot be pickled. So, please use `partial` method from `functools` package instead:

```
from functools import partial

# This is not pickable
model = Sequential(
    L.Convolution2D(None, 64, 3, 1, 1),
    lambda x: F.max_pooling_2d(x, 2)
)

# This is pickable
model = Sequential(
    L.Convolution2D(None, 64, 3, 1, 1),
    partial(F.max_pooling_2d, ksize=2)
)
```

Parameters **layers** – The layers which are called in its order. Each component should be a callable object including *Link* object and functions defined under the *chainer.functions*, e.g., *relu()*, etc.

Methods

__call__(**args*, ***kwargs*)
Call self as a function.

__getitem__(*i*)
Returns the child at given index.

Parameters **index** (*int*) – Index of the child in the list.

Returns The `index`-th child link.

Return type *Link*

__setitem__(*i*, *layer*)

__len__()
Returns the number of children.

__iter__()

add_hook(*hook*, *name=None*)
Registers a link hook.

Parameters

- **hook** (`LinkHook`) – Link hook to be registered.
- **name** (*str*) – Name of the link hook. The name must be unique among link hooks registered to this link. If `None`, the default name of the link hook is used.

Returns self

add_link(*link*)
Registers a child link and adds it to the tail of the list.

Parameters **link** (`Link`) – The link object to be registered.

add_param(*name*, *shape=None*, *dtype=<class 'numpy.float32'>*, *initializer=None*)
Registers a parameter to the link.

Parameters

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** (*initializer*) – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

add_persistent(*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (`Link`) – Source link object.

append(*layer*)
S.append(value) – append value to the end of the sequence

children()
Returns a generator of all child links.

Returns A generator object that generates all child links.

clear() → None – remove all items from S

cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy(*mode='share'*)
Returns a deep copy of the chainlist.

copyparams(*link*, *copy_persistent=True*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

From v5.0.0: this method also copies the persistent values (e.g. the moving statistics of `BatchNormalization`). If the persistent value is an ndarray, the elements are copied. Otherwise, it is copied using `copy.deepcopy()`. The old behavior (not copying persistent values) can be reproduced with `copy_persistent=False`.

Parameters

- **link** (`Link`) – Source link object.
- **copy_persistent** (`bool`) – If `True`, persistent values are also copied. `True` by default.

count(*value*) → integer – return number of occurrences of value

count_by_layer_type(*type_name*)
Count the number of layers by layer type.

This method counts the number of layers which have the name given by the argument `type_name`. For example, if you want to know the number of `Linear` layers included in this model, `type_name` should be `Linear`. If you want to know the number of `Function` classes or user-defined functions which have a specific name, `type_name` should be the function name, e.g., `relu` or `reshape`, etc.

Parameters **type_name** (`str`) – The class or function name of a layer you want to enumerate.

count_params()
Counts the total number of parameters.

This method counts the total number of scalar values included in all the `Parameter`s held by this link and its descendants.

If the link containts uninitialized parameters, this method raises a warning.

Returns The total size of parameters (int)

delete_hook(*name*)
Unregisters the link hook.

Parameters **name** (`str`) – The name of the link hook to be unregistered.

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (`DeviceResidentsVisitor`) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `False`.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

extend(*sequential*)

S.extend(iterable) – extend sequence by appending elements from the iterable

flatten()

Flatten nested `Sequential` links.

This method flattens all the nested `Sequential` links inside this `Sequential` link.

Returns A flattened `Sequential` object.

Example

```
>>> import chainer
>>> import chainer.functions as F
>>> import chainer.links as L
>>> a = chainer.Sequential(L.Linear(None, 10), F.relu)
>>> b = chainer.Sequential(L.Linear(None, 10), F.relu)
>>> a.append(b)
>>> print(a)  # Without flatten
0       Linear  W(None) b(10,)
1       relu
2       Sequential      which has 2 layers
>>> print(a.flatten())  # With flatten
0       Linear  W(None) b(10,)
1       relu
2       Linear  W(None) b(10,)
3       relu
```

---

forward(**x*)

Forward pass computation.

This method performs the forward pass computation by giving the input variable `x` to the layers registered in the constructor in the same order as the order in which the arguments are given to the constructor.

It should be noted that the input variable is given directly to the first layer and all intermediate outputs generated during the forward pass are also directly fed to the next layer. Therefore, the number of outputs at a layer should be the same as the number of inputs at the next layer.

Parameters **x** – Input variables.

Returns The output of the final layer in the given layers.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

init_scope()

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

insert (*i*, *layer*)
Insert a child link at the given index.

Parameters

- **index** (*int*) – The position of the list where the new
- **is inserted.** (*link*) –
- **link** (*Link*) – The link to be inserted.

links (*skipself=False*)
Returns a generator of all links under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

namedlinks (*skipself=False*)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams (*include_uninit=True*)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params (*include_uninit=True*)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit** (*bool*) – If `True`, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

register_persistent (*name*)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters **name** (*str*) – Name of the attribute to be registered.

remove (*layer*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

remove_by_layer_type (*type_name*)
Remove layers by layer type.

This method removes layers from the Sequential object by the layer's class name or function name. If you want to remove a *Link*, the argument `type_name` should be its class name, e.g., *Linear* or *Convolution2D*, etc. If you want to remove a *Function* class or any other callable objects, `type_name` should be the function name, e.g., `relu` or `reshape`, etc.

Parameters **type_name** (*str*) – The name of a layer you want to remove.

repeat (*n_repeat*, *mode='init'*)
Repeats this link multiple times to make a *Sequential*.

This method returns a *Sequential* object which has the same *Link* multiple times repeatedly. The `mode` argument means how to copy this link to repeat.

Example

You can repeat the same link multiple times to create a longer *Sequential* block like this:

```
class ConvBNReLU(chainer.Chain):

    def __init__(self):
        super(ConvBNReLU, self).__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(
                None, 64, 3, 1, 1, nobias=True)
            self.bn = L.BatchNormalization(64)

    def forward(self, x):
        return F.relu(self.bn(self.conv(x)))

net = ConvBNReLU().repeat(16, mode='init')
```

The `net` object contains 16 blocks, each of which is `ConvBNReLU`. And the `mode` was `init`, so each block is re-initialized with different parameters. If you give `copy` to this argument, each block has same values for its parameters but its object ID is different from others. If it is `share`, each block is same to others in terms of not only parameters but also the object IDs because they are shallow-copied, so that when the parameter of one block is changed, all the parameters in the others also change.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned *Sequential* will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting *Sequential* object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

reverse()
S.reverse() – reverse *IN PLACE*

serialize(*serializer*)
Serializes the link object.

Parameters serializer (AbstractSerializer) – Serializer object.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters device – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

Warning: This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters device – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

zerograds()
Initializes all gradient arrays by zero.

Deprecated since version v1.15: Use the more efficient `cleargrads()` instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

__add__(*other*)

Attributes

device
`Device` instance.

local_link_hooks
Ordered dictionary of registered link hooks.

Contrary to `chainer.thread_local.link_hooks`, which registers its elements to all functions, link hooks in this property are specific to this link.

printable_specs
Generator of printable specs of this link.

Yields *specs (tuple of str and object)* – Basically, it returns the arguments (pair of keyword and value) that are passed to the `__init__()`. This pair of key and value is used for representing this class or subclass with `__str__()`.

update_enabled
`True` if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.3.6 Link hooks

Chainer provides a link-hook mechanism that enriches the behavior of `Link`.

`chainer.link_hooks.SpectralNormalization`	Spectral Normalization link hook implementation.
`chainer.link_hooks.TimerHook`	Link hook for measuring elapsed time of `Link.forward()`.
`chainer.link_hooks.WeightStandardization`	Weight Standardization (WS) link hook implementation.

chainer.link_hooks.SpectralNormalization

class chainer.link_hooks.**SpectralNormalization**(*n_power_iteration=1*, *eps=1e-06*, *use_gamma=False*, *factor=None*, *weight_name='W'*, *name=None*)

Spectral Normalization link hook implementation.

This hook normalizes a weight using max singular value and this value is computed via power iteration method. Currently, this hook is supposed to be added to `chainer.links.Linear`, `chainer.links.EmbedID`, `chainer.links.Convolution2D`, `chainer.links.ConvolutionND`, `chainer.links.Deconvolution2D`, and `chainer.links.DeconvolutionND`. However, you can use this to other links like RNNs by specifying `weight_name`. It is highly recommended to add this hook before optimizer setup because this hook add a scaling parameter `gamma` if `use_gamma` is True. Otherwise, the registered `gamma` will not be updated.

$$\bar{\mathbf{W}} = \frac{\mathbf{W}}{\sigma(\mathbf{W})}, \text{ where } \sigma(\mathbf{W}) := \max_{\mathbf{h}: \mathbf{h} \neq 0} \frac{\|\mathbf{W}\mathbf{h}\|_2}{\|\mathbf{h}\|_2} = \max_{\|\mathbf{h}\|_2 \leq 1} \|\mathbf{W}\mathbf{h}\|_2$$

See: T. Miyato et. al., Spectral Normalization for Generative Adversarial Networks

Parameters

- **n_power_iteration** (*int*) – Number of power iteration. The default value is 1.
- **eps** (*float*) – Numerical stability in norm calculation. The default value is 1e-6 for the compatibility with mixed precision training. The value used in the author's implementation is 1e-12.
- **use_gamma** (*bool*) – If `True`, weight scaling parameter gamma which is initialized by initial weight's max singular value is introduced.
- **factor** (*float, None*) – Scaling parameter to divide maximum singular value. The default value is 1.0.
- **weight_name** (*str*) – Link's weight name to apply this hook. The default value is `'W'`.
- **name** (*str or None*) – Name of this hook. The default value is `'SpectralNormalization'`.

Variables

- **vector_name** (*str*) – Name of the approximate first left singular vector registered in the target link. the target link.
- **axis** (*int*) – Axis of weight represents the number of output feature maps or output units (`out_channels` and `out_size`, respectively).

Example

There are almost the same but 2 ways to apply spectral normalization (SN) hook to links.

1. Initialize link and SN separately. This makes it easy to handle buffer and parameter of links registered by SN hook.

```
>>> l = L.Convolution2D(3, 5, 3)
>>> hook = chainer.link_hooks.SpectralNormalization()
>>> _ = l.add_hook(hook)
>>> # Check the shape of the first left singular vector.
>>> getattr(l, hook.vector_name).shape
(5,)
>>> # Delete SN hook from this link.
>>> l.delete_hook(hook.name)
```

2. Initialize both link and SN hook at one time. This makes it easy to define your original `Chain`.

```
>>> # SN hook handles lazy initialization!
>>> layer = L.Convolution2D(
...     5, 3, stride=1, pad=1).add_hook(
...         chainer.link_hooks.SpectralNormalization())
```

Methods

`__enter__`()

`__exit__`()

`added`(*link*)

Callback function invoked when the link hook is registered

Parameters **`link`** (`Link`) – Link object to which the link hook is registered. `None` if the link hook is registered globally.

`deleted`(*link*)

Callback function invoked when the link hook is unregistered

Parameters **`link`** (`Link`) – Link object to which the link hook is unregistered. `None` if the link hook had been registered globally.

`forward_postprocess`(*cb_args*)

Callback function invoked after a forward call of a link.

Parameters **`args`** – Callback data. It has the following attributes:

- **link** (`Link`) Link object.
- **forward_name** (`str`) Name of the forward method.
- **args** (`tuple`) Non-keyword arguments given to the forward method.
- **kwargs** (`dict`) Keyword arguments given to the forward method.
- **out** Return value of the forward method.

`forward_preprocess`(*cb_args*)

Callback function invoked before a forward call of a link.

Parameters **`args`** – Callback data. It has the following attributes:

- **link** (`Link`) Link object.
- **forward_name** (`str`) Name of the forward method.
- **args** (`tuple`) Non-keyword arguments given to the forward method.
- **kwargs** (`dict`) Keyword arguments given to the forward method.

normalize_weight (*link*)
Normalize target weight before every single forward computation.

reshape_W (*W*)
Reshape & transpose weight into 2D if necessary.

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

Attributes

name = 'SpectralNormalization'

chainer.link_hooks.TimerHook

class chainer.link_hooks.**TimerHook**
Link hook for measuring elapsed time of `Link.forward()`.

Example

Code example:

```
from chainer.link_hooks import TimerHook
hook = TimerHook()
with hook:
    trainer.run()
hook.print_report()
```

Output example:

```
  LinkName  ElapsedTime  Occurrence
    Linear     41.42sec        2100
       MLP     42.09sec         700
Classifier     42.39sec         700
```

where *LinkName* is the name of link that calls the hook, and *ElapsedTime* is the elapsed time the link consumed, and *Occurrence* is the number of calls.

Warning: Call graph of links are hierarchical. That means reported elapsed times may be overlapping with each other and the sum may exceed the total time.

Variables **`call_history`** – List of measurement results. It consists of pairs of the name of the link that calls this hook and the elapsed time the `forward()` method of link consumes.

Methods

`__enter__`()

`__exit__`(*_)

`added`(*link*)
Callback function invoked when the link hook is registered

Parameters **`link`** (`Link`) – Link object to which the link hook is registered. `None` if the link hook is registered globally.

`deleted`(*link*)
Callback function invoked when the link hook is unregistered

Parameters **`link`** (`Link`) – Link object to which the link hook is unregistered. `None` if the link hook had been registered globally.

`forward_postprocess`(*args*)
Callback function invoked after a forward call of a link.

Parameters **`args`** – Callback data. It has the following attributes:

- **link (`Link`)** Link object.
- **forward_name (`str`)** Name of the forward method.
- **args (`tuple`)** Non-keyword arguments given to the forward method.
- **kwargs (`dict`)** Keyword arguments given to the forward method.
- **out** Return value of the forward method.

`forward_preprocess`(*args*)
Callback function invoked before a forward call of a link.

Parameters **`args`** – Callback data. It has the following attributes:

- **link (`Link`)** Link object.
- **forward_name (`str`)** Name of the forward method.
- **args (`tuple`)** Non-keyword arguments given to the forward method.
- **kwargs (`dict`)** Keyword arguments given to the forward method.

`print_report`(*unit='auto', file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Prints a summary report of time profiling in links.

Parameters **`unit`** (`str`) – Supplementary units used for computational times. *sec*, *ms*, *us*, *ns*, *auto'(default) and 'auto_foreach* are supported. If *auto*, units of times are aligned to the largest, and if *auto_foreach*, units of times are adjusted for each element.

`summary`()
Returns a summary of time profiling in links.

Returns A summarized dictionary whose keys are link names and values are dictionaries of *elapsed_time* and *occurrence*.

`total_time`()
Returns total elapsed time in seconds.

__eq__()
  Return self==value.

__ne__()
  Return self!=value.

__lt__()
  Return self<value.

__le__()
  Return self<=value.

__gt__()
  Return self>value.

__ge__()
  Return self>=value.

Attributes

name = 'TimerHook'

table = {'ms': 1000, 'ns': 1000000000, 'sec': 1, 'us': 1000000}

chainer.link_hooks.WeightStandardization

class chainer.link_hooks.**WeightStandardization**(*eps=1e-05*, *weight_name='W'*, *name=None*)

Weight Standardization (WS) link hook implementation.

This hook standardizes a weight by *weight statistics*.

This link hook implements a WS which computes the mean and variance along axis "output channels", then normalizes by these statistics. WS improves training by reducing the Lipschitz constants of the loss and the gradients like batch normalization (BN) but without relying on large batch sizes during training. Specifically, the performance of WS with group normalization (GN) trained with small-batch is able to match or outperforms that of BN trained with large-batch. WS is originally proposed for 2D convolution layers followed by mainly GN and sometimes BN. Note that this hook is able to handle layers such as N-dimensional convolutional, linear and embedding layers but there is no guarantee that this hook helps training.

See: Siyuan Qiao et. al., Weight Standardization

Parameters

- **eps** (*float*) – Numerical stability in standard deviation calculation. The default value is 1e-5.
- **weight_name** (*str*) – Link's weight name to appky this hook. The default value is `'W'`.
- **name** (*str or None*) – Name of this hook. The default value is `'WeightStandardization'`.

Methods

__enter__()

__exit__()

added(*link*)
  Callback function invoked when the link hook is registered

Parameters **link** (Link) – Link object to which the link hook is registered. None if the link hook is registered globally.

deleted(*link*)
Callback function invoked when the link hook is unregistered

Parameters **link** (Link) – Link object to which the link hook is unregistered. None if the link hook had been registered globally.

forward_postprocess(*cb_args*)
Callback function invoked after a forward call of a link.

Parameters **args** – Callback data. It has the following attributes:

- **link** (*Link*) Link object.
- **forward_name** (str) Name of the forward method.
- **args** (tuple) Non-keyword arguments given to the forward method.
- **kwargs** (dict) Keyword arguments given to the forward method.
- **out** Return value of the forward method.

forward_preprocess(*cb_args*)
Callback function invoked before a forward call of a link.

Parameters **args** – Callback data. It has the following attributes:

- **link** (*Link*) Link object.
- **forward_name** (str) Name of the forward method.
- **args** (tuple) Non-keyword arguments given to the forward method.
- **kwargs** (dict) Keyword arguments given to the forward method.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'WeightStandardization'

You can also implement your own link-hook to inject arbitrary code before/after the forward propagation.

chainer.LinkHook	Base class of hooks for links.

chainer.LinkHook

class chainer.**LinkHook**

Base class of hooks for links.

LinkHook is a callback object that is registered to a *Link*. Registered link hooks are invoked before and after calling `Link.forward()` method of each link.

Link hooks that derive from *LinkHook* may override the following method:

- *added()*
- *deleted()*
- *forward_preprocess()*
- *forward_postprocess()*

By default, these methods do nothing.

Specifically, when the *__call__()* method of some link is invoked, *forward_preprocess()* (resp. *forward_postprocess()*) of all link hooks registered to this link are called before (resp. after) `Link.forward()` method of the link.

There are two ways to register *LinkHook* objects to *Link* objects.

The first one is to use `with` statement. Link hooks hooked in this way are registered to all links within `with` statement and are unregistered at the end of `with` statement.

Example

The following code is a simple example in which we measure the elapsed time of a part of forward propagation procedure with *TimerHook*, which is a subclass of *LinkHook*.

```
>>> class Model(chainer.Chain):
...   def __init__(self):
...     super(Model, self).__init__()
...     with self.init_scope():
...       self.l = L.Linear(10, 10)
...   def forward(self, x1):
...     return F.exp(self.l(x1))
>>> model1 = Model()
>>> model2 = Model()
>>> x = chainer.Variable(np.zeros((1, 10), np.float32))
>>> with chainer.link_hooks.TimerHook() as m:
...   _ = model1(x)
...   y = model2(x)
>>> model3 = Model()
>>> z = model3(y)
>>> print('Total time : {}'.format(m.total_time()))
...
Total time : ...
```

In this example, we measure the elapsed times for each forward propagation of all functions in `model1` and `model2`. Note that `model3` is not a target measurement as *TimerHook* is unregistered before forward propagation of `model3`.

Note: Chainer stores the dictionary of registered link hooks as a thread local object. So, link hooks registered are different depending on threads.

The other one is to register directly to a *Link* object by calling its *add_hook()* method. Link hooks registered in this way can be removed by *delete_hook()* method. Contrary to former registration method, link hooks are registered only to the link which *add_hook()* is called.

Parameters **name** (*str*) – Name of this link hook.

Methods

__enter__()

__exit__(*_)

added(*link*)
Callback function invoked when the link hook is registered

Parameters **link** (`Link`) – Link object to which the link hook is registered. `None` if the link hook is registered globally.

deleted(*link*)
Callback function invoked when the link hook is unregistered

Parameters **link** (`Link`) – Link object to which the link hook is unregistered. `None` if the link hook had been registered globally.

forward_postprocess(*args*)
Callback function invoked after a forward call of a link.

Parameters **args** – Callback data. It has the following attributes:

- **link** (*Link*) Link object.
- **forward_name** (`str`) Name of the forward method.
- **args** (`tuple`) Non-keyword arguments given to the forward method.
- **kwargs** (`dict`) Keyword arguments given to the forward method.
- **out** Return value of the forward method.

forward_preprocess(*args*)
Callback function invoked before a forward call of a link.

Parameters **args** – Callback data. It has the following attributes:

- **link** (*Link*) Link object.
- **forward_name** (`str`) Name of the forward method.
- **args** (`tuple`) Non-keyword arguments given to the forward method.
- **kwargs** (`dict`) Keyword arguments given to the forward method.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = 'LinkHook'

4.4 Probability Distributions

Chainer provides many `Distribution` implementations in the `chainer.distributions` package.

4.4.1 Distributions

`chainer.distributions.Bernoulli`	Bernoulli Distribution.
`chainer.distributions.Beta`	Beta Distribution.
`chainer.distributions.Categorical`	Categorical Distribution.
`chainer.distributions.Cauchy`	Cauchy Distribution.
`chainer.distributions.Chisquare`	Chi-Square Distribution.
`chainer.distributions.Dirichlet`	Dirichlet Distribution.
`chainer.distributions.Exponential`	Exponential Distribution.
`chainer.distributions.Gamma`	Gamma Distribution.
`chainer.distributions.Geometric`	Geometric Distribution.
`chainer.distributions.Gumbel`	Gumbel Distribution.
`chainer.distributions.Independent`	Independent distribution.
`chainer.distributions.Laplace`	Laplace Distribution.
`chainer.distributions.LogNormal`	Logatithm Normal Distribution.
`chainer.distributions.MultivariateNormal`	MultivariateNormal Distribution.
`chainer.distributions.Normal`	Normal Distribution.
`chainer.distributions.OneHotCategorical`	OneHotCategorical Distribution.
`chainer.distributions.Pareto`	Pareto Distribution.
`chainer.distributions.Poisson`	Poisson Distribution.
`chainer.distributions.Uniform`	Uniform Distribution.

chainer.distributions.Bernoulli

class chainer.distributions.**Bernoulli**(*p=None*, *logit=None*, *binary_check=False*)
Bernoulli Distribution.

The probability mass function of the distribution is expressed as

$$P(x = 1; p) = p$$
$$P(x = 0; p) = 1 - p$$

Parameters

- **p** (`Variable` or *N-dimensional array*) – Parameter of distribution representing p. Either p or *logit* (not both) must have a value.
- **logit** (`Variable` or *N-dimensional array*) – distribution representing $\log\{p/(1-p)\}$. Either *p* or *logit* (not both) must have a value.

Methods

cdf (*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf (*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf (*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob (*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function (*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity (*x*)

Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy
Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

logit

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

p

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Beta

class chainer.distributions.**Beta**(*a*, *b*)

Beta Distribution.

The probability density function of the distribution is expressed as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)},$$

for $0 < x < 1$, $\alpha > 0$, $\beta > 0$.

Parameters

- **a** (*Variable* or *N-dimensional array*) – Parameter of distribution representing α.
- **b** (*Variable* or *N-dimensional array*) – Parameter of distribution representing β.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

a

b

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp

Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Categorical

class chainer.distributions.**Categorical**(*p=None*, ***kwargs*)

Categorical Distribution.

The probability mass function of the distribution is expressed as

$$P(x = i; p) = p_i$$

Parameters

- **p** (*Variable* or *N-dimensional array*) – Parameter of distribution.
- **logit** (*Variable* or *N-dimensional array*) – Parameter of distribution representing $\log\{p\} + C$. Either *p* or *logit* (not both) must have a value.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

log_p

mean
Returns the mean of the distribution.

Returns The mean of the distribution.

Return type *Variable*

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

p

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance
Returns the variance of the distribution.

Returns The variance of the distribution.

Return type *Variable*

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Cauchy

class `chainer.distributions.`**`Cauchy`**(*loc*, *scale*)
Cauchy Distribution.

The probability density function of the distribution is expressed as

$$p(x; x_0, \gamma) = \frac{1}{\pi} \frac{\gamma}{(x - x_0)^2 + \gamma^2}$$

Parameters

- **loc** (*Variable* or *N-dimensional array*) – Parameter of distribution representing the location x_0.
- **scale** (*Variable* or *N-dimensional array*) – Parameter of distribution representing the scale γ.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)

Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)

Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

batch_shape

Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance

Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

loc

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Chisquare

class `chainer.distributions.`**`Chisquare`**`(k)`
Chi-Square Distribution.

The probability density function of the distribution is expressed as

$$p(x; k) = \frac{1}{2^{k/2}\Gamma(k/2)}x^{k/2-1}e^{-x/2}$$

Parameters **k** (`Variable` or *N-dimensional array*) – Parameter of distribution.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

k

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Dirichlet

class `chainer.distributions.`**`Dirichlet`**(*alpha*)
Dirichlet Distribution.

The probability density function of the distribution is expressed as

$$p(x) = \frac{\Gamma(\sum_{i=1}^{K} \alpha_i)}{\prod_{i=1}^{K} \Gamma(\alpha_i)} \prod_{i=1}^{K} {x_i}^{\alpha_i - 1}$$

Parameters **alpha** (`Variable` or *N-dimensional array*) – Parameter of distribution.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)

Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)

Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)

Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)

Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

alpha

alpha0

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Exponential

class chainer.distributions.**Exponential**(*lam*)

Exponential Distribution.

The probability density function of the distribution is expressed as

$$p(x;\lambda) = \lambda e^{-\lambda x}$$

Parameters **lam** (*Variable* or *N-dimensional array*) – Parameter of distribution λ.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

lam

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp

Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Gamma

class `chainer.distributions.`**Gamma**(*k*, *theta*)

Gamma Distribution.

Parameters

- **k** (*Variable* or *N-dimensional array*) – Parameter of distribution.
- **theta** (*Variable* or *N-dimensional array*) – Parameter of distribution.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

k

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

theta

variance

xp

Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Geometric

class `chainer.distributions.`**`Geometric`**(*p*)

Geometric Distribution.

The probability mass function of the distribution is expressed as

$$Pr(x = k) = p(1 - p)^{k-1}, for k = 1, 2, 3, ...,$$

Parameters **p** (`Variable` or *N-dimensional array*) – Parameter of distribution.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy
Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

p

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Gumbel

class `chainer.distributions.Gumbel`(*loc*, *scale*)
Gumbel Distribution.

The probability density function of the distribution is expressed as

$$f(x) = \frac{1}{\eta} \exp\left\{ - \frac{x - \mu}{\eta} \right\} \exp\left[- \exp\left\{ - \frac{x - \mu}{\eta} \right\} \right],$$

Parameters

- **loc** (`Variable` or *N-dimensional array*) – Parameter of distribution μ.
- **scale** (`Variable` or *N-dimensional array*) – Parameter of distribution η.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

loc

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Independent

class `chainer.distributions.`**`Independent`**(*distribution*, *reinterpreted_batch_ndims=None*)
Independent distribution.

Parameters

- **distribution** (*Distribution*) – The base distribution instance to transform.
- **reinterpreted_batch_ndims** (`int`) – Integer number of rightmost batch dims which will be regarded as event dims. When `None` all but the first batch axis (batch axis 0) will be transferred to event dimensions.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
The inverse cumulative distribution function for multivariate variable.

Cumulative distribution function for multivariate variable is not invertible. This function always raises `RuntimeError`.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the codomain of the distribution

Raises `RuntimeError` –

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,) + batch_shape + event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
The covariance of the independent distribution.

By definition, the covariance of the new distribution becomes block diagonal matrix. Let $\Sigma_{\mathbf{x}}$ be the covariance matrix of the original random variable $\mathbf{x} \in \mathbb{R}^d$, and $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots \mathbf{x}^{(m)}$ be the m i.i.d. random variables, new covariance matrix $\Sigma_{\mathbf{y}}$ of $\mathbf{y} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(m)}] \in \mathbb{R}^{md}$ can be written as

$$\begin{bmatrix} \Sigma_{\mathbf{x}^1} & & 0 \\ & \ddots & \\ 0 & & \Sigma_{\mathbf{x}^m} \end{bmatrix}.$$

Note that this relationship holds only if the covariance matrix of the original distribution is given analytically.

Returns The covariance of the distribution.

Return type *Variable*

distribution

entropy
Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean
Returns the mean of the distribution.

Returns The mean of the distribution.

Return type *Variable*

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

reinterpreted_batch_ndims

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance
Returns the variance of the distribution.

Returns The variance of the distribution.

Return type *Variable*

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Laplace

class `chainer.distributions.`**`Laplace`**(*loc*, *scale*)

Laplace Distribution.

The probability density function of the distribution is expressed as

$$p(x; \mu, b) = \frac{1}{2b} \exp \left(-\frac{|x - \mu|}{b} \right)$$

Parameters

- **loc** (*Variable* or *N-dimensional array*) – Parameter of distribution representing the location μ.
- **scale** (*Variable* or *N-dimensional array*) – Parameter of distribution representing the scale b.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

loc

mean

mode

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.LogNormal

class `chainer.distributions.`**`LogNormal`**(*mu*, *sigma*)
Logatithm Normal Distribution.

The probability density function of the distribution is expressed as

$$p(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}x} \exp\left(-\frac{(\log x-\mu)^2}{2\sigma^2}\right)$$

Parameters

- **mu** (*Variable* or *N-dimensional array*) – Parameter of distribution μ.
- **sigma** (*Variable* or *N-dimensional array*) – Parameter of distribution σ.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters x (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

mu

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

sigma

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support

Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp

Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.MultivariateNormal

class `chainer.distributions.`**`MultivariateNormal`**(*loc*, ***kwargs*)

MultivariateNormal Distribution.

The probability density function of the distribution is expressed as

$$p(x;\mu,V) = \frac{1}{\sqrt{\det(2\pi V)}} \exp\left(-\frac{1}{2}(x-\mu)V^{-1}(x-\mu)\right)$$

Parameters

- **loc** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the location μ.
- **scale_tril** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the scale L such that $V = LL^T$.

Methods

__copy__()

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

d

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

loc

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale_tril

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance
Returns the variance of the distribution.

Returns The variance of the distribution.

Return type *Variable*

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Normal

class `chainer.distributions.`**`Normal`**(*loc*, *scale=None*, ***kwargs*)
Normal Distribution.

The probability density function of the distribution is expressed as

$$p(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Parameters

- **loc** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the location μ. This is the mean parameter.
- **scale** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the scale σ. Either *scale* or *log_scale* (not both) must have a value.
- **log_scale** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the scale $\log(\sigma)$. Either *scale* or *log_scale* (not both) must have a value.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)

Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)

Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

batch_shape

Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance

Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

loc

log_scale

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.OneHotCategorical

class `chainer.distributions.`**`OneHotCategorical`**(*p*)
OneHotCategorical Distribution.

Parameters **p** (*Variable* or *N-dimensional array*) – Parameter of distribution.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)

Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)

Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

batch_shape

Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance

Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

log_p

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

p

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Pareto

class `chainer.distributions.`**`Pareto`**(*scale*, *alpha*)
Pareto Distribution.

$$f(x) = \alpha x_m^{\alpha}(x)^{-(\alpha+1)},$$

Parameters

- **scale** (`Variable` or *N-dimensional array*) – Parameter of distribution x_m.
- **alpha** (`Variable` or *N-dimensional array*) – Parameter of distribution α.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)

Evaluates the perplexity function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)

Evaluates probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)

Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)

Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)

Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

alpha

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Poisson

class chainer.distributions.**Poisson**(*lam*)

Poisson Distribution.

The probability mass function of the distribution is expressed as

$$P(x; \lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$$

Parameters **lam** (*Variable* or *N-dimensional array*) – Parameter of distribution. λ

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)

Evaluates the logarithm of survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy
Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

lam

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev
Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp

Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

chainer.distributions.Uniform

class `chainer.distributions.`**`Uniform`**`(**kwargs)`

Uniform Distribution.

The probability density function of the distribution is expressed as

$$p(x; l, h) = \begin{cases} \frac{1}{h-l} & \text{if } l \leq x \leq h \\ 0 & \text{otherwise} \end{cases}$$

Parameters

- **low** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the lower bound l.
- **high** (`Variable` or *N-dimensional array*) – Parameter of distribution representing the higher bound h.

Methods

cdf(*x*)

Evaluates the cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)

Evaluates the inverse cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)

Evaluates the log of cumulative distribution function at the given points.

Parameters x (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)

Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (`tuple` of `int`) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (`int`) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance
Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

event_shape
Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

high

loc

low

mean

mode
Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params
Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

scale

stddev

support
Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

4.4.2 Functionals of distribution

`chainer.cross_entropy`	Computes Cross entropy.
`chainer.kl_divergence`	Computes Kullback-Leibler divergence.
`chainer.register_kl`	Decorator to register KL divergence function.

chainer.cross_entropy

`chainer.cross_entropy(dist1, dist2)`
Computes Cross entropy.

For two continuous distributions $p(x), q(x)$, it is expressed as

$$H(p,q) = -\int p(x) \log q(x) dx$$

For two discrete distributions $p(x), q(x)$, it is expressed as

$$H(p,q) = -\sum_x p(x) \log q(x)$$

This function call `kl_divergence()` and `entropy()` of `dist1`. Therefore, it is necessary to register KL divergence function with `register_kl()` decoartor and define `entropy()` in `dist1`.

Parameters

- **dist1** (`Distribution`) – Distribution to calculate cross entropy p. This is the first (left) operand of the cross entropy.
- **dist2** (`Distribution`) – Distribution to calculate cross entropy q. This is the second (right) operand of the cross entropy.

Returns Output variable representing cross entropy $H(p,q)$.

Return type *Variable*

chainer.kl_divergence

`chainer.kl_divergence(dist1, dist2)`
Computes Kullback-Leibler divergence.

For two continuous distributions $p(x), q(x)$, it is expressed as

$$D_{KL}(p||q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

For two discrete distributions $p(x), q(x)$, it is expressed as

$$D_{KL}(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

Parameters

- **dist1** (*Distribution*) – Distribution to calculate KL divergence p. This is the first (left) operand of the KL divergence.
- **dist2** (*Distribution*) – Distribution to calculate KL divergence q. This is the second (right) operand of the KL divergence.

Returns Output variable representing kl divergence $D_{KL}(p||q)$.

Return type *Variable*

Using `register_kl()`, we can define behavior of `kl_divergence()` for any two distributions.

chainer.register_kl

`chainer.register_kl(Dist1, Dist2)`

Decorator to register KL divergence function.

This decorator registers a function which computes Kullback-Leibler divergence. This function will be called by `kl_divergence()` based on the argument types.

Parameters

- **Dist1** (*type*) – type of a class inherit from `Distribution` to calculate KL divergence.
- **Dist2** (*type*) – type of a class inherit from `Distribution` to calculate KL divergence.

The decorated functoion takes an instance of `Dist1` and `Dist2` and returns KL divergence value.

Example

This is a simple example to register KL divergence. A function to calculate a KL divergence value between an instance of `Dist1` and an instance of `Dist2` is registered.

```
from chainer import distributions
@distributions.register_kl(Dist1, Dist2)
def _kl_dist1_dist2(dist1, dist2):
    return KL
```

4.4.3 Base classes

`chainer.Distribution`	Interface of Distribution

chainer.Distribution

class chainer.**Distribution**

Interface of Distribution

Distribution is a bass class for dealing with probability distributions.

This class provides the following capabilities.

1. Sampling random points.
2. Evaluating a probability-related function at a given realization value. (e.g., probability density function, probability mass function)
3. Obtaining properties of distributions. (e.g., mean, variance)

Note that every method and property that computes them from *chainer.Variable* can basically be differentiated.

In this class, sampled random points and realization values given in probability-related function is called *sample*. Sample consists of *batches*, and each batch consists of independent *events*. Each event consists of values, and each value in an event cannot be sampled independently in general. Each event in a batch is independent while it is not sampled from an identical distribution. And each batch in sample is sampled from an identical distribution.

Each part of the sample-batch-event hierarchy has its own shape, which is called `sample_shape`, `batch_shape`, and `event_shape`, respectively.

On initialization, it takes distribution-specific parameters as inputs. *batch_shape* and *event_shape* is decided by the shape of the parameter when generating an instance of a class.

Example

The following code is an example of sample-batch-event hierarchy on using *MultivariateNormal* distribution. This makes 2d normal distributions. `dist` consists of 12(4 * 3) independent 2d normal distributions. And on initialization, *batch_shape* and *event_shape* is decided.

```
>>> import chainer
>>> import chainer.distributions as D
>>> import numpy as np
>>> d = 2
>>> shape = (4, 3)
>>> loc = np.random.normal(
...     size=shape + (d,)).astype(np.float32)
>>> cov = np.random.normal(size=shape + (d, d)).astype(np.float32)
>>> cov = np.matmul(cov, np.rollaxis(cov, -1, -2))
>>> l = np.linalg.cholesky(cov)
>>> dist = D.MultivariateNormal(loc, scale_tril=l)
>>> dist.event_shape
(2,)
>>> dist.batch_shape
(4, 3)
>>> sample = dist.sample(sample_shape=(6, 5))
>>> sample.shape
(6, 5, 4, 3, 2)
```

Every probability-related function takes realization value whose shape is the concatenation of `sample_shape`, `batch_shape`, and `event_shape` and returns an evaluated value whose shape is the concatenation of `sample_shape`, and `batch_shape`.

Methods

cdf(*x*)
Evaluates the cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Cumulative distribution function value evaluated at *x*.

Return type *Variable*

icdf(*x*)
Evaluates the inverse cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Inverse cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_cdf(*x*)
Evaluates the log of cumulative distribution function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of cumulative distribution function value evaluated at *x*.

Return type *Variable*

log_prob(*x*)
Evaluates the logarithm of probability at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of probability evaluated at *x*.

Return type *Variable*

log_survival_function(*x*)
Evaluates the logarithm of survival function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Logarithm of survival function value evaluated at *x*.

Return type *Variable*

perplexity(*x*)
Evaluates the perplexity function at the given points.

Parameters **x** (`Variable` or *N-dimensional array*) – Data points in the domain of the distribution

Returns Perplexity function value evaluated at *x*.

Return type *Variable*

prob(*x*)
Evaluates probability at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Probability evaluated at *x*.

Return type *Variable*

sample(*sample_shape=()*)
Samples random points from the distribution.

This function calls *sample_n* and reshapes a result of *sample_n* to *sample_shape* + *batch_shape* + *event_shape*. On implementing sampling code in an inherited distribution class, it is not recommended that you override this function. Instead of doing this, it is preferable to override *sample_n*.

Parameters **sample_shape** (tuple of int) – Sampling shape.

Returns Sampled random points.

Return type *Variable*

sample_n(*n*)
Samples n random points from the distribution.

This function returns sampled points whose shape is *(n,)* + *batch_shape* + *event_shape*. When implementing sampling code in a subclass, it is recommended that you override this method.

Parameters **n** (*int*) – Sampling size.

Returns sampled random points.

Return type *Variable*

survival_function(*x*)
Evaluates the survival function at the given points.

Parameters **x** (*Variable* or *N-dimensional array*) – Data points in the domain of the distribution

Returns Survival function value evaluated at *x*.

Return type *Variable*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_shape
Returns the shape of a batch.

Returns The shape of a sample that is not identical and independent.

Return type tuple

covariance

Returns the covariance of the distribution.

Returns The covariance of the distribution.

Return type *Variable*

entropy

Returns the entropy of the distribution.

Returns The entropy of the distribution.

Return type *Variable*

event_shape

Returns the shape of an event.

Returns The shape of a sample that is not identical and independent.

Return type tuple

mean

Returns the mean of the distribution.

Returns The mean of the distribution.

Return type *Variable*

mode

Returns the mode of the distribution.

Returns The mode of the distribution.

Return type *Variable*

params

Returns the parameters of the distribution.

Returns The parameters of the distribution.

Return type dict

stddev

Returns the standard deviation of the distribution.

Returns The standard deviation of the distribution.

Return type *Variable*

support

Returns the support of the distribution.

Returns String that means support of this distribution.

Return type str

variance

Returns the variance of the distribution.

Returns The variance of the distribution.

Return type *Variable*

xp
Array module for the distribution.

Depending on which of CPU/GPU this distribution is on, this property returns `numpy` or `cupy`.

4.5 Optimizers

chainer.optimizers.AdaDelta	Zeiler's ADADELTA.
chainer.optimizers.AdaGrad	AdaGrad optimizer.
chainer.optimizers.Adam	Adam optimizer.
chainer.optimizers.AdamW	AdamW optimizer.
chainer.optimizers.AMSGrad	AMSGrad optimizer.
chainer.optimizers.AdaBound	AdaBound optimizer.
chainer.optimizers.AMSBound	AMSBound optimizer.
chainer.optimizers.CorrectedMomentumSGD	Momentum SGD optimizer.
chainer.optimizers.MomentumSGD	Momentum SGD optimizer.
chainer.optimizers.NesterovAG	Nesterov's Accelerated Gradient.
chainer.optimizers.MSVAG	M-SVAG optimizer.
chainer.optimizers.RMSprop	RMSprop optimizer.
chainer.optimizers.RMSpropGraves	Alex Graves's RMSprop.
chainer.optimizers.SGD	Vanilla Stochastic Gradient Descent.
chainer.optimizers.SMORMS3	Simon Funk's SMORMS3.

4.5.1 chainer.optimizers.AdaDelta

class chainer.optimizers.**AdaDelta**(*rho=0.95*, *eps=1e-06*)
Zeiler's ADADELTA.

See: http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf

Parameters

- **rho** (*float*) – Exponential decay rate of the first and second order moments.
- **eps** (*float*) – Small value for the numerical stability.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.

- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale (*loss_scale*)
Sets loss scaling factor.

setup (*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update (*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale ()

use_cleargrads (*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update (*flag=True*)
Enables use of parameter update in fp32.

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__()

Return self>=value.

Attributes

epoch = 0

eps

Alias to `self.hyperparam.eps`

rho

Alias to `self.hyperparam.rho`

t = 0

target = None

use_auto_new_epoch = False

4.5.2 chainer.optimizers.AdaGrad

class chainer.optimizers.**AdaGrad**(*lr=0.001*, *eps=1e-08*)

AdaGrad optimizer.

See: http://jmlr.org/papers/v12/duchi11a.html

Parameters

- **lr** (*float*) – Learning rate.
- **eps** (*float*) – Small value for the numerical stability.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)

Invokes hook functions in registration order.

check_nan_in_grads()

Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by *setup()* to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If None, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters auto (*bool*) – Should be True if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to True by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have None. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters name (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

eps
Alias to `self.hyperparam.eps`

lr
Alias to `self.hyperparam.lr`

t = 0

target = None

use_auto_new_epoch = False

4.5.3 chainer.optimizers.Adam

class chainer.optimizers.**Adam**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *eps=1e-08*, *eta=1.0*, *weight_decay_rate=0*, *amsgrad=False*, *adabound=False*, *final_lr=0.1*, *gamma=0.001*)

Adam optimizer.

See: Adam: A Method for Stochastic Optimization

Modified for proper weight decay (also called *AdamW*). AdamW introduces the additional parameters `eta` and `weight_decay_rate`, which can be used to properly scale the learning rate, and decouple the weight decay rate from `alpha`, as shown in the below paper.

Note that with the default values `eta = 1` and `weight_decay_rate = 0`, this implementation is identical to the standard Adam method.

See: Fixing Weight Decay Regularization in Adam

A flag `amsgrad` to use the *AMSGrad* variant of Adam from the paper: On the Convergence of Adam and Beyond

A flag `adabound` to use the *AdaBound* variant of Adam from the paper: Adaptive Gradient Methods with Dynamic Bound of Learning Rate

If both `amsgrad` and `adabound` are `True`, the optimizer is equivalent to *AMSBound* proposed in the AdaBound paper.

Parameters

- **alpha** (*float*) – Coefficient of learning rate.
- **beta1** (*float*) – Exponential decay rate of the first order moment.
- **beta2** (*float*) – Exponential decay rate of the second order moment.
- **eps** (*float*) – Small value for the numerical stability.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts.
- **weight_decay_rate** (*float*) – Weight decay rate.
- **amsgrad** (*bool*) – Whether to use AMSGrad variant of Adam.
- **adabound** (*bool*) – Whether to use the AdaBound variant of Adam.
- **final_lr** (*float*) – Final (SGD) learning rate in AdaBound.
- **gamma** (*float*) – Convergence speed of the bound functions in AdaBound.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters link (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters use (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

adabound
Alias to `self.hyperparam.adabound`

alpha
Alias to `self.hyperparam.alpha`

alpha_t

amsgrad
Alias to `self.hyperparam.amsgrad`

beta1
Alias to `self.hyperparam.beta1`

beta2
Alias to `self.hyperparam.beta2`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eta
Alias to `self.hyperparam.eta`

final_lr
Alias to `self.hyperparam.final_lr`

gamma
Alias to `self.hyperparam.gamma`

lr

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.4 chainer.optimizers.AdamW

class chainer.optimizers.**AdamW**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *eps=1e-08*, *eta=1.0*, *weight_decay_rate=0*)
AdamW optimizer.

This class is a special case of `Adam`.

See: Fixing Weight Decay Regularization in Adam

Parameters

- **alpha** (*float*) – Coefficient of learning rate.
- **beta1** (*float*) – Exponential decay rate of the first order moment.
- **beta2** (*float*) – Exponential decay rate of the second order moment.
- **eps** (*float*) – Small value for the numerical stability.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts. The default value is 1.0.
- **weight_decay_rate** (*float*) – Weight decay rate. The default value is 0.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters use (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

adabound
Alias to `self.hyperparam.adabound`

alpha
Alias to `self.hyperparam.alpha`

alpha_t

amsgrad
Alias to `self.hyperparam.amsgrad`

beta1
Alias to `self.hyperparam.beta1`

beta2
Alias to `self.hyperparam.beta2`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eta
Alias to `self.hyperparam.eta`

final_lr
Alias to `self.hyperparam.final_lr`

gamma
Alias to `self.hyperparam.gamma`

lr

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.5 chainer.optimizers.AMSGrad

class `chainer.optimizers.`**AMSGrad**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *eps=1e-08*, *eta=1.0*)
AMSGrad optimizer.

This class is a special case of `Adam`.

See: On the Convergence of Adam and Beyond

Parameters

- **alpha** (*float*) – Coefficient of learning rate.
- **beta1** (*float*) – Exponential decay rate of the first order moment.
- **beta2** (*float*) – Exponential decay rate of the second order moment.
- **eps** (*float*) – Small value for the numerical stability.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the `target` attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

adabound
Alias to `self.hyperparam.adabound`

alpha
Alias to `self.hyperparam.alpha`

alpha_t

amsgrad
Alias to `self.hyperparam.amsgrad`

beta1
Alias to `self.hyperparam.beta1`

beta2
Alias to `self.hyperparam.beta2`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eta
Alias to `self.hyperparam.eta`

final_lr
Alias to `self.hyperparam.final_lr`

gamma
Alias to `self.hyperparam.gamma`

lr

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.6 chainer.optimizers.AdaBound

class `chainer.optimizers.`**AdaBound**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *final_lr=0.1*, *gamma=0.001*, *eps=1e-08*, *eta=1.0*)

AdaBound optimizer.

This class is a special case of `Adam`.

See: Adaptive Gradient Methods with Dynamic Bound of Learning Rate

Parameters

- **alpha** (*float*) – Coefficient of learning rate.
- **beta1** (*float*) – Exponential decay rate of the first order moment.
- **beta2** (*float*) – Exponential decay rate of the second order moment.
- **final_lr** (*float*) – Final (SGD) learning rate in AdaBound.
- **gamma** (*float*) – Convergence speed of the bound functions in AdaBound.
- **eps** (*float*) – Small value for the numerical stability.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (`str`) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

adabound
Alias to `self.hyperparam.adabound`

alpha
Alias to `self.hyperparam.alpha`

alpha_t

amsgrad
Alias to `self.hyperparam.amsgrad`

beta1
Alias to `self.hyperparam.beta1`

beta2
Alias to `self.hyperparam.beta2`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eta
Alias to `self.hyperparam.eta`

final_lr
Alias to `self.hyperparam.final_lr`

gamma
Alias to `self.hyperparam.gamma`

lr

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.7 chainer.optimizers.AMSBound

class chainer.optimizers.**AMSBound**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *final_lr=0.1*, *gamma=0.001*, *eps=1e-08*, *eta=1.0*)

AMSBound optimizer.

This class is a special case of *Adam*.

See: Adaptive Gradient Methods with Dynamic Bound of Learning Rate

Parameters

- **alpha** (*float*) – Coefficient of learning rate.
- **beta1** (*float*) – Exponential decay rate of the first order moment.
- **beta2** (*float*) – Exponential decay rate of the second order moment.
- **final_lr** (*float*) – Final (SGD) learning rate in AdaBound.
- **gamma** (*float*) – Convergence speed of the bound functions in AdaBound.
- **eps** (*float*) – Small value for the numerical stability.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)

Invokes hook functions in registration order.

check_nan_in_grads()

Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()

Creates a new update rule object.

This method creates an update rule object. It is called by *setup()* to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the `target` attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (`bool`) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

adabound
Alias to `self.hyperparam.adabound`

alpha
Alias to `self.hyperparam.alpha`

alpha_t

amsgrad
Alias to `self.hyperparam.amsgrad`

beta1
Alias to `self.hyperparam.beta1`

beta2
Alias to `self.hyperparam.beta2`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eta
Alias to `self.hyperparam.eta`

final_lr
Alias to `self.hyperparam.final_lr`

gamma
Alias to `self.hyperparam.gamma`

lr

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.8 chainer.optimizers.CorrectedMomentumSGD

class chainer.optimizers.**CorrectedMomentumSGD**(*lr=0.01*, *momentum=0.9*)
Momentum SGD optimizer.

This implements momentum correction discussed in the third section of Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.

`MomentumSGD` implements the equation (10) of the paper. This optimizer implements the equation (9).

To get better understanding between the two methods, we show the equivalence between the equation (9) and modification of the equation (10) that takes momentum correction into account. First, we set $v_t = \eta_t u_t$. We substitute this relation to the equation (10).

$$\begin{aligned} v_{t+1} &= m\frac{\eta_{t+1}}{\eta_t}v_t + \eta_{t+1}g_t \\ &= m\frac{\eta_{t+1}}{\eta_t}\eta_t u_t + \eta_{t+1}g_t \\ &= \eta_{t+1}(mu_t + g_t) \end{aligned}$$

From this result, we derive $u_{t+1} = mu_t + g_t$, which is how update tensors are calculated by `CorrectedMomentumSGD`. Thus, the equivalence is shown.

Parameters

- **lr** (*float*) – Learning rate.
- **momentum** (*float*) – Exponential decay rate of the first order moment.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (AbstractSerializer) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

lr
Alias to `self.hyperparam.lr`

momentum
Alias to `self.hyperparam.momentum`

t = 0

target = None

use_auto_new_epoch = False

4.5.9 chainer.optimizers.MomentumSGD

class chainer.optimizers.**MomentumSGD**(*lr=0.01*, *momentum=0.9*)
Momentum SGD optimizer.

Parameters

- **lr** (*float*) – Learning rate.
- **momentum** (*float*) – Exponential decay rate of the first order moment.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (`int`) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (`float`) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (`bool`) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (`str`) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

lr
Alias to `self.hyperparam.lr`

momentum
Alias to `self.hyperparam.momentum`

t = 0

target = None

use_auto_new_epoch = False

4.5.10 chainer.optimizers.NesterovAG

class `chainer.optimizers.`**NesterovAG**(*lr=0.01*, *momentum=0.9*)
Nesterov's Accelerated Gradient.

See: https://arxiv.org/abs/1212.0901

Parameters

- **lr** (*float*) – Learning rate.
- **momentum** (*float*) – Exponential decay rate of the first order moment.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the `target` attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

lr
Alias to `self.hyperparam.lr`

momentum
Alias to `self.hyperparam.momentum`

t = 0

target = None

use_auto_new_epoch = False

4.5.11 chainer.optimizers.MSVAG

class `chainer.optimizers.`**`MSVAG`**(*lr=0.1*, *beta=0.9*, *eta=1.0*, *weight_decay_rate=0*)

M-SVAG optimizer.

See: Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients

Modified for proper weight decay (also called AdamW). AdamW introduces the additional parameters `eta` and `weight_decay_rate`, which can be used to properly scale the learning rate, and decouple the weight decay rate from `alpha`, as shown in the below paper.

See: Fixing Weight Decay Regularization in Adam

Parameters

- **lr** (*float*) – Learning rate.
- **beta** (*float*) – Exponential decay rate of the first and second order moment.
- **eta** (*float*) – Schedule multiplier, can be used for warm restarts.
- **weight_decay_rate** (*float*) – Weight decay rate.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)

Invokes hook functions in registration order.

check_nan_in_grads()

Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()

Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

beta
Alias to `self.hyperparam.beta`

epoch = 0

eta
Alias to `self.hyperparam.eta`

lr
Alias to `self.hyperparam.lr`

t = 0

target = None

use_auto_new_epoch = False

weight_decay_rate
Alias to `self.hyperparam.weight_decay_rate`

4.5.12 chainer.optimizers.RMSprop

class chainer.optimizers.**RMSprop**(*lr=0.01*, *alpha=0.99*, *eps=1e-08*, *eps_inside_sqrt=False*)

RMSprop optimizer.

See: T. Tieleman and G. Hinton (2012). Lecture 6.5 - rmsprop, COURSERA: Neural Networks for Machine Learning.

Parameters

- **lr** (*float*) – Learning rate.
- **alpha** (*float*) – Exponential decay rate of the second order moment.
- **eps** (*float*) – Small value for the numerical stability.
- **eps_inside_sqrt** (*bool*) – When `True`, gradient will be divided by $\sqrt{ms + eps}$ where `ms` is the mean square. When `False` (default), gradient will be divided by $\sqrt{ms} + eps$ instead. This option may be convenient for users porting code from other frameworks; see #4754 for details.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)

Invokes hook functions in registration order.

check_nan_in_grads()

Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()

Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If None, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters auto (*bool*) – Should be True if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to True by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have None. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters name (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters link (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

alpha
Alias to `self.hyperparam.alpha`

epoch = 0

eps
Alias to `self.hyperparam.eps`

eps_inside_sqrt
Alias to `self.hyperparam.eps_inside_sqrt`

lr
Alias to `self.hyperparam.lr`

t = 0

target = None

use_auto_new_epoch = False

4.5.13 chainer.optimizers.RMSpropGraves

class chainer.optimizers.**RMSpropGraves**(*lr=0.0001*, *alpha=0.95*, *momentum=0.9*, *eps=0.0001*)

Alex Graves's RMSprop.

See: https://arxiv.org/abs/1308.0850

Parameters

- **lr** (*float*) – Learning rate.
- **alpha** (*float*) – Exponential decay rate of the first and second order moments of the raw gradient.
- **momentum** (*float*) – Exponential decay rate of the first order moment of the adjusted gradient.
- **eps** (*float*) – Small value for the numerical stability.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)

Invokes hook functions in registration order.

check_nan_in_grads()

Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()

Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)

Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)

Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters auto (*bool*) – Should be `True` if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to `True` by the updater.

reallocate_cleared_grads()

Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)

Removes a hook function.

Parameters name (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)

Sets loss scaling factor.

setup(*link*)

Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters link (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.

- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of `cleargrads()` in *update*.

Parameters use (`bool`) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

alpha
Alias to `self.hyperparam.alpha`

epoch = 0

eps
Alias to `self.hyperparam.eps`

lr
Alias to `self.hyperparam.lr`

momentum
Alias to `self.hyperparam.momentum`

t = 0

target = None

use_auto_new_epoch = False

4.5.14 chainer.optimizers.SGD

class chainer.optimizers.**SGD**(*lr=0.01*)
Vanilla Stochastic Gradient Descent.

Parameters **lr** (*float*) – Learning rate.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **`auto`** (*`bool`*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

`reallocate_cleared_grads`()

Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

`remove_hook`(*name*)

Removes a hook function.

Parameters **`name`** (*`str`*) – Registered name of the hook function to remove.

`serialize`(*serializer*)

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **`serializer`** (`AbstractSerializer`) – Serializer or deserializer object.

`set_loss_scale`(*loss_scale*)

Sets loss scaling factor.

`setup`(*link*)

Sets a target link and initializes the optimizer states.

Given link is set to the `target` attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **`link`** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

`update`(*lossfun=None*, **args*, ***kwds*)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

`update_loss_scale`()

`use_cleargrads`(*use=True*)

Enables or disables use of `cleargrads()` in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

use_fp32_update (*flag=True*)
Enables use of parameter update in fp32.

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

Attributes

epoch = 0

lr
Alias to `self.hyperparam.lr`

t = 0

target = None

use_auto_new_epoch = False

4.5.15 chainer.optimizers.SMORMS3

class chainer.optimizers.**SMORMS3** (*lr=0.001*, *eps=1e-16*)
Simon Funk's SMORMS3.

See http://sifter.org/~simon/journal/20150420.html.

Parameters

- **lr** (*float*) – Learning rate.
- **eps** (*float*) – Small value for the numerical stability.

Methods

add_hook (*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000, scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, `use_auto_new_epoch` should be set to `True` by the updater.

reallocate_cleared_grads()
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (AbstractSerializer) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (Link) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)
Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

eps
Alias to `self.hyperparam.eps`

lr
Alias to `self.hyperparam.lr`

t = 0

target = None

use_auto_new_epoch = False

4.5.16 Optimizer base classes

chainer.Optimizer	Base class of all numerical optimizers.
chainer.UpdateRule	Base class of all update rules.
chainer.optimizer.Hyperparameter	Set of hyperparameter entries of an optimizer.
chainer.GradientMethod	Base class of all single gradient-based optimizers.

chainer.Optimizer

class chainer.**Optimizer**
Base class of all numerical optimizers.

This class provides basic features for all optimization methods. It optimizes parameters of a *target link*. The target link is registered via the *setup()* method, and then the *update()* method updates its parameters based on a given loss function.

Each optimizer implementation must be defined as a child class of Optimizer. It must override *update()* method.

If the optimizer is based on single gradient computation (like most first-order methods), then it should inherit *GradientMethod*, which adds some features dedicated for the first order methods, including the support of *UpdateRule*.

Optimizer instance also supports *hook functions*. Hook function is registered by the *add_hook()* method. Each hook function is called in registration order before of after the actual parameter update (configurable). If the hook function has an attribute `call_for_each_param` and its value is `True`, the hook function is used as a hook function of all update rules (i.e., it is invoked for every parameter by passing the corresponding update rule and the parameter).

Variables

- ***target*** – Target link object. It is set by the *setup()* method.
- ***t*** – Number of update steps. It must be incremented by the *update()* method.
- ***epoch*** – Current epoch. It is incremented by the *new_epoch()* method.
- ***use_auto_new_epoch*** – Boolean flag to indicate if *new_epoch()* will be called by the updater. Updater should set this flag to `True` if it automatically calls *new_epoch()*.

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the *epoch* count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to `True` by the updater.

remove_hook(*name*)
Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)
Sets loss scaling factor.

setup(*link*)
Sets a target link and initializes the optimizer states.

Given link is set to the `target` attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

---

update(*lossfun=None*, **args*, ***kwds*)
Updates the parameters.

This method updates the parameters of the target link. The behavior of this method is different for the cases either `lossfun` is given or not.

If `lossfun` is given, this method typically clears the gradients, calls the loss function with given extra arguments, and calls the `backward()` method of its output to compute the gradients. The actual implementation might call `lossfun` more than once.

If `lossfun` is not given, then this method assumes that the gradients of all parameters are already computed. An implementation that requires multiple gradient computations might raise an error on this case.

In both cases, this method invokes the update procedure for all parameters.

Parameters

- **lossfun** (`callable`) – Loss function. You can specify one of loss functions from *built-in loss functions*, or your own loss function. It should not be an *loss functions with parameters* (i.e., `Link` instance). The function must accept arbitrary arguments and return one `Variable` object that represents the loss (or objective) value. Returned value must be a Variable derived from the input Variable object. `lossfun` can be omitted for single gradient-based methods. In this case, this method assumes gradient arrays computed.
- **kwds** (`args,`) – Arguments for the loss function.

update_loss_scale()

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

t = 0

target = None

use_auto_new_epoch = False

chainer.UpdateRule

class chainer.**UpdateRule**(*parent_hyperparam=None*)
Base class of all update rules.

Update rule is an object that implements how to update one parameter variable using the gradient of a loss function. This class provides the interface and the common features of any update rules.

An update rule can be set to a `Variable` object that represents a parameter array of a model. An `Optimizer` instance defines which parameters to update, and the update rule instance of each parameter defines how to update it.

Hook functions can be set to any update rule instance. The hook function is called just before or after any updates (configurable) in the order of registrations.

An implementation of update rule should override `update_core()` or its device-dependent variants (i.e., `update_core_cpu()` and `update_core_gpu()`).

The state (e.g. a moving average of the gradient) of the update rule is stored into the state dictionary. An implementation of update rule using state should also override `init_state()` to initialize the state at the first update. The values of the state dictionary are automatically copied to the appropriate device before the update based on the data and grad arrays.

Parameters parent_hyperparam (`Hyperparameter`) – Hyperparameter that provides the default values.

Variables

- **enabled** (`bool`) – Flag to configure if this update rule is active. If the update rule is not active (i.e., `enabled = False`), the `update()` method does not update the parameter.
- **hyperparam** (`Hyperparameter`) – Hyperparameter of the update rule.
- **t** (`int`) – Number of updates made by this update rule.

Methods

add_hook (*hook*, *name=None*, *timing='auto'*)
Adds a hook function.

The hook function is called before or after any updates (see the timing attribute).

Parameters

- **hook** (`callable`) – Hook function to be added. It takes two arguments: the update rule object and the parameter variable.
- **name** (`str`) – Name of the hook function. The name attribute of the hook function is used by default.
- **timing** (`str`) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates. If 'auto' and the timing property of the hook is not available, timing will default to 'pre'.

init_state (*param*)
Initializes the state.

Any implementations that use the state should override this mehtod. This method is called at the first update.

Parameters **param** (`Variable`) – Parameter variable. It can be used to extract the shape and the data type of the parameter.

remove_hook (*name*)
Removes the specified hook function.

Parameters **name** (`str`) – Name of the hook function to be removed. The hook function registered with this name will be removed.

serialize (*serializer*)
Serializes the update rule state.

Be careful that this method only saves/loads the state of the update rule. The parameters of the target link is not saved/loaded by this method, and so you need to serialize the target link separately if you want to fully recover the training state including parameters.

Parameters **serializer** (`AbstractSerializer`) – Serializer object.

update (*param*)
Invokes hook functions and updates the parameter.

Parameters **param** (`Variable`) – Variable to be updated.

update_core (*param*)
Updates the parameter.

Implementation of UpdateRule should override this method or both of `update_core_cpu()` and `update_core_gpu()`.

Parameters **param** (`Variable`) – Variable to be updated.

update_core_chainerx (*param*)
Updates the ChainerX parameter.

This method can be overridden to implement custom update logic. The default implementation is to convert the parameter to a memory-shared NumPy/CuPy parameter and call the corresponding update method.

See `update_core()` for details.

Parameters **param** (Variable) – Variable to be updated.

update_core_cpu(*param*)
Updates the parameter on CPU.

See *update_core()* for details.

Parameters **param** (Variable) – Variable to be updated.

update_core_gpu(*param*)
Updates the parameter on GPU.

See *update_core()* for details.

Parameters **param** (Variable) – Variable to be updated.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

This method enables use of parameter update in fp32. When it is enabled and data type of original parameter variable is fp16, fp32 copy of parameter variable is automatically created and retained at self.fp32_param. And the parameter is update in fp32 in the following way.

1. copies the grad of original parameter variable to the grad of fp32 parameter variable, converting its data type from fp16 to fp32.
2. updates the parameter in fp32.
3. copies the data of fp32 parameter variable to the data of original parameter variable, converting its data type from fp32 to fp16.

See *update()* for details.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

state
State dictionary.

chainer.optimizer.Hyperparameter

class chainer.optimizer.**Hyperparameter**(*parent=None*)
Set of hyperparameter entries of an optimizer.

This is a utility class to provide a set of hyperparameter entries for update rules and an optimizer. Each entry can be set as an attribute of a hyperparameter object.

A hyperparameter object can hold a reference to its parent hyperparameter object. When an attribute does not exist in the child hyperparameter, it automatically refers to the parent. We typically set the hyperparameter of the gradient method as the parent of the hyperparameter of each update rule. It enables us to centralize the management of hyperparameters (e.g. we can change the learning rate of all update rules just by modifying the hyperparameter of the central optimizer object), while users can freely customize the hyperparameter of each update rule if needed.

Parameters **parent** (`Hyperparameter`) – Parent hyperparameter.

Methods

get_dict()

Converts the hyperparameter into a dictionary.

Returns Dictionary containing all entries that can be referred by this hyperparameter object.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

parent

Parent hyperparameter object.

chainer.GradientMethod

class chainer.**GradientMethod**

Base class of all single gradient-based optimizers.

This is an extension of the *Optimizer* class. Typical gradient methods that just require the gradient at the current parameter vector on an update can be implemented as its child class.

This class uses *UpdateRule* to manage the update rule of each parameter. A child class of GradientMethod should override *create_update_rule()* to create the default update rule of each parameter.

This class also provides `hyperparam`, which is the hyperparameter used as the default configuration of each update rule. All built-in gradient method implementations also provide proxy properties that act as aliases to the attributes of `hyperparam`. It is recommended that you provide such an alias to each attribute. It can be done by only adding one line for each attribute using `HyperparameterProxy`.

Variables **hyperparam** (Hyperparameter) – The hyperparameter of the gradient method. It is used as the default configuration of each update rule (i.e., the hyperparameter of each update rule refers this hyperparameter as its parent).

Methods

add_hook(*hook*, *name=None*, *timing='auto'*)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method, and the timing attribute.

Parameters

- **hook** (*callable*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
- **timing** (*str*) – Specifies when the hook is called. If 'auto', the timimg property of the hook will decide the timing. If 'pre', the hook will be called before any updates. If 'post', the hook will be called after any updates.

call_hook(*hook*)

call_hooks(*timing='pre'*)
Invokes hook functions in registration order.

check_nan_in_grads()
Checks if there is NaN in grads when dynamic loss scaling used.

create_update_rule()
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type *UpdateRule*

is_safe_to_update()

loss_scaling(*interval=1000*, *scale=None*)
Configures the loss scaling algorithm.

Parameters

- **interval** (*int*) – Number of iterations until scaling factor gets doubled. This is effective when "dynamic" loss scaling is used.
- **scale** (*float*) – Loss scaling factor. If `None`, "dynamic" loss scaling is used, otherwise "static" loss scaling is used.

new_epoch(*auto=False*)
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

Parameters **auto** (*bool*) – Should be `True` if this method is called by an updater. In this case, *use_auto_new_epoch* should be set to `True` by the updater.

reallocate_cleared_grads()

Reallocate gradients cleared by *cleargrad()*.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

remove_hook(*name*)

Removes a hook function.

Parameters **name** (*str*) – Registered name of the hook function to remove.

serialize(*serializer*)

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (*t* and *epoch*)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters **serializer** (`AbstractSerializer`) – Serializer or deserializer object.

set_loss_scale(*loss_scale*)

Sets loss scaling factor.

setup(*link*)

Sets a target link and initializes the optimizer states.

Given link is set to the *target* attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters **link** (`Link`) – Target link object.

Returns The optimizer instance.

Note: As of v4.0.0, this function returns the optimizer instance itself so that you can instantiate and setup the optimizer in one line, e.g., `optimizer = SomeOptimizer().setup(link)`.

update(*lossfun=None*, **args*, ***kwds*)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

update_loss_scale()

use_cleargrads(*use=True*)

Enables or disables use of *cleargrads()* in *update*.

Parameters **use** (*bool*) – If `True`, this function enables use of *cleargrads*. If `False`, disables use of *cleargrads* (*zerograds* is used).

Deprecated since version v2.0: Note that *update()* calls *cleargrads()* by default. *cleargrads()* is more efficient than *zerograds()*, so one does not have to call *use_cleargrads()*. This method remains for backward compatibility.

use_fp32_update(*flag=True*)
Enables use of parameter update in fp32.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

epoch = 0

t = 0

target = None

use_auto_new_epoch = False

4.5.17 Hook functions

chainer.optimizer_hooks.WeightDecay	Optimizer/UpdateRule hook function for weight decay regularization.
chainer.optimizer_hooks.Lasso	Optimizer/UpdateRule hook function for Lasso regularization.
chainer.optimizer_hooks.GradientClipping	Optimizer hook function for gradient clipping.
chainer.optimizer_hooks.GradientHardClipping	Optimizer/UpdateRule hook function for gradient clipping.
chainer.optimizer_hooks.GradientNoise	Optimizer/UpdateRule hook function for adding gradient noise.
chainer.optimizer_hooks.GradientLARS	Optimizer/UpdateRule hook function for layer wise adaptive rate scaling.

chainer.optimizer_hooks.WeightDecay

class chainer.optimizer_hooks.**WeightDecay**(*rate*)
Optimizer/UpdateRule hook function for weight decay regularization.

This hook function adds a scaled parameter to the corresponding gradient. It can be used as a regularization.

Parameters **rate** (*float*) – Coefficient for the weight decay.

Variables

- **rate** (*float*) – Coefficient for the weight decay.
- ***timing*** (*string*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).
- ***call_for_each_param*** (*bool*) – Specifies if this hook is called for each parameter (`True`) or only once (`False`) by an optimizer to which this hook is registered. This function does not expect users to switch the value from default one, which is *True*.

New in version 4.0.0: The *timing* parameter.

Methods

__call__(*rule, param*)
Call self as a function.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

call_for_each_param = True

name = 'WeightDecay'

timing = 'pre'

chainer.optimizer_hooks.Lasso

class chainer.optimizer_hooks.**Lasso**(*rate*)
Optimizer/UpdateRule hook function for Lasso regularization.

This hook function adds a scaled parameter to the sign of each weight. It can be used as a regularization.

Parameters **rate** (*float*) – Coefficient for the weight decay.

Variables

- **rate** (*float*) – Coefficient for the weight decay.

- ***timing*** (*string*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).
- ***call_for_each_param*** (*bool*) – Specifies if this hook is called for each parameter (True) or only once (False) by an optimizer to which this hook is registered. This function does not expect users to switch the value from default one, which is *True*.

New in version 4.0.0: The *timing* parameter.

Methods

__call__(*rule*, *param*)
Call self as a function.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

call_for_each_param = True

name = 'Lasso'

timing = 'pre'

chainer.optimizer_hooks.GradientClipping

class chainer.optimizer_hooks.**GradientClipping**(*threshold*)
Optimizer hook function for gradient clipping.

This hook function scales all gradient arrays to fit to the defined L2 norm threshold.

Parameters **threshold** (*float*) – L2 norm threshold.

Variables

- **threshold** (*float*) – L2 norm threshold of gradient norm.
- ***timing*** (*string*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).

New in version 4.0.0: The *timing* parameter.

Methods

`__call__`(*opt*)
Call self as a function.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Attributes

`name = 'GradientClipping'`

`timing = 'pre'`

chainer.optimizer_hooks.GradientHardClipping

class `chainer.optimizer_hooks.`**`GradientHardClipping`**(*lower_bound*, *upper_bound*)
Optimizer/UpdateRule hook function for gradient clipping.

This hook function clips all gradient arrays to be within a lower and upper bound.

Parameters

- **`lower_bound`** (*`float`*) – The lower bound of the gradient value.
- **`upper_bound`** (*`float`*) – The upper bound of the gradient value.

Variables

- **`lower_bound`** (*`float`*) – The lower bound of the gradient value.
- **`upper_bound`** (*`float`*) – The upper bound of the gradient value.
- ***`timing`*** (*`string`*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).
- ***`call_for_each_param`*** (*`bool`*) – Specifies if this hook is called for each parameter (`True`) or only once (`False`) by an optimizer to which this hook is registered. This function does not expect users to switch the value from default one, which is *True*.

New in version 4.0.0: The *timing* parameter.

Methods

__call__(*rule*, *param*)
Call self as a function.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

call_for_each_param = True

name = 'GradientHardClipping'

timing = 'pre'

chainer.optimizer_hooks.GradientNoise

class chainer.optimizer_hooks.**GradientNoise**(*eta*, *noise_func=<function exponential_decay_noise>*)
Optimizer/UpdateRule hook function for adding gradient noise.

This hook function simply adds noise generated by the `noise_func` to the gradient. By default it adds time-dependent annealed Gaussian noise to the gradient at every training step:

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

where

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

with η selected from $\{0.01, 0.3, 1.0\}$ and $\gamma = 0.55$.

Parameters

- **eta** (*float*) – Parameter that defines the scale of the noise. For the default noise function, it is recommended that it be either 0.01, 0.3 or 1.0.
- **noise_func** (*function*) – Noise generating function which by default is given by Adding Gradient Noise Improves Learning for Very Deep Networks.

Variables

- ***timing*** (*string*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).

- **call_for_each_param** (*bool*) – Specifies if this hook is called for each parameter (`True`) or only once (`False`) by an optimizer to which this hook is registered. This function does not expect users to switch the value from default one, which is *True*.

New in version 4.0.0: The *timing* parameter.

Methods

__call__(*rule*, *param*)
Call self as a function.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

call_for_each_param = True

name = 'GradientNoise'

timing = 'pre'

chainer.optimizer_hooks.GradientLARS

class chainer.optimizer_hooks.**GradientLARS**(*threshold=0.01*, *weight_decay=0.0*, *eps=1e-09*)

Optimizer/UpdateRule hook function for layer wise adaptive rate scaling.

See: Large Batch Training of Convolutional Networks.

See: Convergence Analysis of Gradient Descent Algorithms with Proportional Updates.

This hook function scales all gradient arrays to fit to the weight norm.

In <https://arxiv.org/abs/1708.03888>,

$$
\begin{aligned}
v_{t+1} &= m * v_t + \gamma * \lambda * (\nabla L(w_t) + \beta w_t), \\
w_{t+1} &= w_t - v_{t+1},
\end{aligned}
$$

where

- γ : learning_rate
- m : momentum

- β : weight_decay
- η : lars_coeeficient
- λ: local_lr $= \eta * \frac{\|w_t\|}{\|\nabla L(w_t)\| + \beta * \|w_t\|}$.

As lr in chainer.optimizers.SGD or chainer.optimizers.MomentumSGD corresponds to $\gamma * \eta$, we define $clip\_rate$ as $\frac{\|w_t\|}{\|\nabla L(w_t)\| + \beta * \|w_t\|}$ and reformulate the aforementioned formula as: $v_{t+1} = m * v_t + lr * clip\_rate * (\nabla L(w_t) + \beta w_t)$ and implement in this way. So you do not set lars_coeeficient.

Parameters

- **threshold** (*float*) – If weight norm is more than threshold, this function scales all gradient arrays to fit weight norm. (See <https://arxiv.org/abs/1801.03137>)
- **weight_decay** (*float*) – Coefficient for the weight decay.
- **eps** (*float*) – Small value for the numerical stability. (See <https://arxiv.org/abs/1801.03137>)

Variables

- **threshold** (*float*) – If weight norm is more than threshold, this function scales all gradient arrays to fit weight norm. (See <https://arxiv.org/abs/1801.03137>)
- **weight_decay** (*float*) – Coefficient for the weight decay.
- **eps** (*float*) – Small value for the numerical stability. (See <https://arxiv.org/abs/1801.03137>)
- ***timing*** (*string*) – Specifies when this hook should be called by the Optimizer/UpdateRule. Valid values are 'pre' (before any updates) and 'post' (after any updates).
- ***call_for_each_param*** (*bool*) – Specifies if this hook is called for each parameter (`True`) or only once (`False`) by an optimizer to which this hook is registered. This function does not expect users to switch the value from default one, which is *True*.

Methods

__call__(*rule*, *param*)
Call self as a function.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

call_for_each_param = True

name = 'GradientLARS'

timing = 'pre'

4.6 Weight Initializers

Weight initializers are used to initialize arrays. They destructively modify the content of `numpy.ndarray` or `cupy.ndarray`. Typically, weight initializers are passed to *Link*s to initialize their weights and biases.

A weight initializer can be any of the following objects.

- *chainer.Initializer* class instance.
- Python or NumPy scalar or `numpy.ndarray`.
- A callable that takes an array (`numpy.ndarray` or `cupy.ndarray`) and feeds the initial data into it.
- `None`, in which case *the default initializer* is used. Unless explicitly documented, it is *LeCunNormal* with scale value 1.

If an initializer object has the `dtype` attribute, the initializer can assume that the array to feed the data into has that dtype. If the required dtype, depending on the context where the initializer is used, does not match the `dtype` attribute, Chainer will report an error.

4.6.1 Base class

chainer.Initializer	Initializes array.

chainer.Initializer

class chainer.**Initializer**(*dtype=None*)

Initializes array.

It initializes the given array.

Variables **dtype** – Data type specifier. It is for type check in `__call__` function.

Methods

__call__(*array*)

Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.6.2 Concrete initializers

chainer.initializers.Identity	Initializes array with the identity matrix.
chainer.initializers.Constant	Initializes array with constant value.
chainer.initializers.Zero	Initializes array to all-zero.
chainer.initializers.One	Initializes array to all-one.
chainer.initializers.NaN	Initializes array to all-NaN.
chainer.initializers.Normal	Initializes array with a normal distribution.
chainer.initializers.LeCunNormal	Initializes array with scaled Gaussian distribution.
chainer.initializers.GlorotNormal	Initializes array with scaled Gaussian distribution.
chainer.initializers.HeNormal	Initializes array with scaled Gaussian distribution.
chainer.initializers.Orthogonal	Initializes array with an orthogonal system.
chainer.initializers.Uniform	Initializes array with a scaled uniform distribution.
chainer.initializers.LeCunUniform	Initializes array with a scaled uniform distribution.
chainer.initializers.GlorotUniform	Initializes array with a scaled uniform distribution.
chainer.initializers.HeUniform	Initializes array with scaled uniform distribution.
chainer.initializers.UpsamplingDeconvFilter	Initializes array with upsampling filter.
chainer.initializers.DownsamplingConvFilter	Initializes array with downsampling filter.

chainer.initializers.Identity

class chainer.initializers.**Identity**(*scale=1.0*, *dtype=None*)
Initializes array with the identity matrix.

It initializes the given array with the constant multiple of the identity matrix. Note that arrays to be passed must be 2D squared matrices.

Variables **scale** (*scalar*) – A constant to be multiplied to identity matrices.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.Constant

class chainer.initializers.**Constant**(*fill_value*, *dtype=None*)
Initializes array with constant value.

Variables

- **fill_value** (scalar or *N-dimensional array*) – A constant to be assigned to the initialized array. Broadcast is allowed on this assignment.
- **dtype** – Data type specifier.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

fill_value = None

chainer.initializers.Zero

class chainer.initializers.**Zero**(*dtype=None*)

Initializes array to all-zero.

Variables **dtype** – Data type specifier.

Methods

__call__(*array*)

Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

fill_value = 0.0

chainer.initializers.One

class chainer.initializers.**One**(*dtype=None*)

Initializes array to all-one.

Variables **dtype** – Data type specifier.

Methods

__call__(*array*)

Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()

Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

fill_value = 1.0

chainer.initializers.NaN

class `chainer.initializers.`**NaN**(*dtype=None*)
Initializes array to all-NaN.

Variables **dtype** – Data type specifier.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

fill_value = nan

chainer.initializers.Normal

class `chainer.initializers.`**`Normal`**(*scale=0.05*, *dtype=None*, ***kwargs*)

Initializes array with a normal distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is `scale`.

Parameters

- **scale** (*float*) – Standard deviation of Gaussian distribution.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

`__call__`(*array*)

Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

`__eq__`()

Return self==value.

`__ne__`()

Return self!=value.

`__lt__`()

Return self<value.

`__le__`()

Return self<=value.

`__gt__`()

Return self>value.

`__ge__`()

Return self>=value.

chainer.initializers.LeCunNormal

class `chainer.initializers.`**`LeCunNormal`**(*scale=1.0*, *dtype=None*, ***kwargs*)

Initializes array with scaled Gaussian distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is $scale \times \sqrt{\frac{1}{fan_{in}}}$, where fan_{in} is the number of input units.

Reference: LeCun 98, Efficient Backprop http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Parameters

- **scale** (*float*) – A constant that determines the scale of the standard deviation.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.GlorotNormal

class chainer.initializers.**GlorotNormal**(*scale=1.0*, *dtype=None*, ***kwargs*)
Initializes array with scaled Gaussian distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is $scale \times \sqrt{\frac{2}{fan_{in}+fan_{out}}}$, where fan_{in} and fan_{out} are the number of input and output units, respectively.

Reference: Glorot & Bengio, AISTATS 2010

Parameters

- **scale** (*float*) – A constant that determines the scale of the standard deviation.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.HeNormal

class `chainer.initializers.`**`HeNormal`**(*scale=1.0*, *dtype=None*, *fan_option='fan_in'*, ***kwargs*)

Initializes array with scaled Gaussian distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is $scale \times \sqrt{\frac{2}{fan}}$. If `fan_option == 'fan_in'`, fan is the number of input units. If `fan_option == 'fan_out'`, fan is the number of output units.

Reference: He et al., https://arxiv.org/abs/1502.01852

Parameters

- **scale** (*float*) – A constant that determines the scale of the standard deviation.
- **dtype** – Data type specifier.
- **fan_option** (*{'fan_in', 'fan_out'}*) – Decides how to compute the standard deviation. The default value is `'fan_in'`.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

`__gt__()`
Return self>value.

`__ge__()`
Return self>=value.

chainer.initializers.Orthogonal

class `chainer.initializers.`**`Orthogonal`**(*scale=1.1*, *dtype=None*, *mode='auto'*, ***kwargs*)

Initializes array with an orthogonal system.

This initializer first makes a matrix of the same shape as the array to be initialized whose elements are drawn independently from standard Gaussian distribution. Next, it applies QR decomposition to (the transpose of) the matrix. To make the decomposition (almost surely) unique, we require the diagonal of the triangular matrix R to be non-negative (see e.g. Edelman & Rao, https://web.eecs.umich.edu/~rajnrao/Acta05rmt.pdf). Then, it initializes the array with the (semi-)orthogonal matrix Q. Finally, the array is multiplied by the constant `scale`.

If the `ndim` of the input array is more than 2, we consider the array to be a matrix by concatenating all axes except the first one.

The number of vectors consisting of the orthogonal system (i.e. first element of the shape of the array) must be equal to or smaller than the dimension of each vector (i.e. second element of the shape of the array).

Variables

- **scale** (*float*) – A constant to be multiplied by.
- **dtype** – Data type specifier.
- **mode** (*str*) – Assertion on the initialized shape. `'auto'` (default), `'projection'` (before v7), `'embedding'`, or `'basis'`.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Reference: Saxe et al., https://arxiv.org/abs/1312.6120

Methods

`__call__`(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

`__eq__()`
Return self==value.

`__ne__()`
Return self!=value.

`__lt__()`
Return self<value.

`__le__()`
Return self<=value.

`__gt__()`
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.Uniform

class chainer.initializers.**Uniform**(*scale=0.05*, *dtype=None*, ***kwargs*)
Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-scale, scale]$.

Variables

- **scale** (*float*) – A constant that determines the scale of the uniform distribution.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.LeCunUniform

class chainer.initializers.**LeCunUniform**(*scale=1.0*, *dtype=None*, ***kwargs*)
Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-s, s]$ where $s = scale \times \sqrt{\frac{3}{fan_{in}}}$. Here fan_{in} is the number of input units.

Reference: LeCun 98, Efficient Backprop http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Variables

- **scale** (*float*) – A constant that determines the scale of the uniform distribution.

- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.GlorotUniform

class chainer.initializers.**GlorotUniform**(*scale=1.0*, *dtype=None*, ***kwargs*)
Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-s, s]$ where $s = scale \times \sqrt{\frac{6}{fan_{in}+fan_{out}}}$. Here, fan_{in} and fan_{out} are the number of input and output units, respectively.

Variables

- **scale** (*float*) – A constant that determines the scale of the uniform distribution.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.HeUniform

class chainer.initializers.**HeUniform**(*scale=1.0*, *dtype=None*, ***kwargs*)
Initializes array with scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-s, s]$ where $s = scale \times \sqrt{\frac{6}{fan_{in}}}$. Here, fan_{in} is the number of input units.

Variables

- **scale** (*float*) – A constant that determines the scale of the uniform distribution.
- **dtype** – Data type specifier.
- **rng** (*xp.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.initializers.UpsamplingDeconvFilter

class chainer.initializers.**UpsamplingDeconvFilter**(*interpolation='linear'*, *dtype=None*)

Initializes array with upsampling filter.

The array is initialized with a standard image upsampling weight. This initializer is often used as inital weight for `DeconvolutionND()`. `DeconvolutionND()` is expected that its *stride* is equal to *(ksize + 1) // 2*.

Reference: Long et al., https://arxiv.org/abs/1411.4038

Variables

- **interpolation** (*str*) – Upsampling interpolation method.
- **is 'linear'.** (*Default*) –

Methods

__call__(*array*)

Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

chainer.initializers.DownsamplingConvFilter

class chainer.initializers.**DownsamplingConvFilter**(*interpolation='linear'*, *dtype=None*)

Initializes array with downsampling filter.

The array is initialized with a standard image downsampling weight. This initializer is often used as inital weight for `ConvolutionND()`. `ConvolutionND()` is expected that its *stride* is equal to *(ksize + 1) // 2*.

Reference: Long et al., https://arxiv.org/abs/1411.4038

Variables

- **interpolation** (*str*) – Downsampling interpolation method.
- **is 'linear'.** (*Default*) –

Methods

__call__(*array*)
Initializes given array.

This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

Parameters **array** (*N-dimensional array*) – An array to be initialized by this initializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.6.3 Helper function

`chainer.initializers.generate_array`	Return initialized array.

chainer.initializers.generate_array

`chainer.initializers.`**generate_array**(*initializer, shape, xp, dtype=None, device=None*)
Return initialized array.

The algorithms used to make the new values depend on the concrete derived classes. If the initializer has the `dtype` attribute, it is used to construct the array. Otherwise, `chainer.config.dtype` is used instead. See *Configuring Chainer* for the dtype config.

Parameters

- **initializer** – A callable object that takes *N-dimensional array* and edits its value.
- **shape** (*int or tuple of int*) – Shape of the initialized array.
- **xp** (*module*) – `cupy`, `numpy`, or `chainerx`.
- **dtype** – Dtype specifier. If omitted, `initializer.dtype` is used.
- **device** – Target device specifier. If omitted, the current device is used for `cupy`, and the default device is used for `chainerx`.

Returns An initialized array.

Return type *N-dimensional array*

4.7 Snapshot Writers

chainer.training.extensions.snapshot_writers.Writer	Base class of snapshot writers.
chainer.training.extensions.snapshot_writers.SimpleWriter	The most simple snapshot writer.
chainer.training.extensions.snapshot_writers.ThreadWriter	Snapshot writer that uses a separate thread.
chainer.training.extensions.snapshot_writers.ProcessWriter	Snapshot writer that uses a separate process.
chainer.training.extensions.snapshot_writers.QueueWriter	Base class of queue snapshot writers.
chainer.training.extensions.snapshot_writers.ThreadQueueWriter	Snapshot writer that uses a thread queue.
chainer.training.extensions.snapshot_writers.ProcessQueueWriter	Snapshot writer that uses process queue.

4.7.1 chainer.training.extensions.snapshot_writers.Writer

class `chainer.training.extensions.snapshot_writers.`**`Writer`**

Base class of snapshot writers.

`Snapshot` invokes `__call__` of this class everytime when taking a snapshot. This class determines how the actual saving function will be invoked.

See also:

- *chainer.training.extensions.snapshot()*

Methods

`__call__`(*filename*, *outdir*, *target*)

Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

`finalize`()

Finalizes the wirter.

Like extensions in *Trainer*, this method is invoked at the end of the training.

`save`(*filename*, *outdir*, *target*, *savefun*, ***kwds*)

`__eq__`()

Return self==value.

`__ne__`()

Return self!=value.

`__lt__()`
Return self<value.

`__le__()`
Return self<=value.

`__gt__()`
Return self>value.

`__ge__()`
Return self>=value.

4.7.2 chainer.training.extensions.snapshot_writers.SimpleWriter

class chainer.training.extensions.snapshot_writers.**SimpleWriter**(*savefun=<function save_npz>*, ***kwds*)

The most simple snapshot writer.

This class just passes the arguments to the actual saving function.

Parameters

- **savefun** – Callable object. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **kwds** – Keyword arguments for the `savefun`.

See also:

- `chainer.training.extensions.snapshot()`

Methods

`__call__`(*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

`finalize()`
Finalizes the wirter.

Like extensions in `Trainer`, this method is invoked at the end of the training.

`save`(*filename*, *outdir*, *target*, *savefun*, ***kwds*)

`__eq__()`
Return self==value.

`__ne__()`
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

4.7.3 chainer.training.extensions.snapshot_writers.ThreadWriter

class `chainer.training.extensions.snapshot_writers.`**`ThreadWriter`**(*savefun=<function save_npz>*, ***kwds*)

Snapshot writer that uses a separate thread.

This class creates a new thread that invokes the actual saving function.

See also:

- `chainer.training.extensions.snapshot()`

Methods

`__call__`(*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (`str`) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (`str`) – Output directory. Corresponds to `Trainer.out`.
- **target** (`dict`) – Serialized object which will be saved.

`create_worker`(*filename*, *outdir*, *target*, ***kwds*)
Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods.

Parameters

- **filename** (`str`) – Name of the file into which the serialized target is saved. It is already formated string.
- **outdir** (`str`) – Output directory. Passed by *trainer.out*.
- **target** (`dict`) – Serialized object which will be saved.
- **kwds** – Keyword arguments for the `savefun`.

`finalize`()
Finalizes the wirter.

Like extensions in `Trainer`, this method is invoked at the end of the training.

save (*filename*, *outdir*, *target*, *savefun*, ***kwds*)

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

4.7.4 chainer.training.extensions.snapshot_writers.ProcessWriter

class `chainer.training.extensions.snapshot_writers.`**ProcessWriter** (*savefun=<function save_npz>*, ***kwds*)

Snapshot writer that uses a separate process.

This class creates a new process that invokes the actual saving function.

Note: Forking a new process from a MPI process might be danger. Consider using *ThreadWriter* instead of `ProcessWriter` if you are using MPI.

See also:

- *chainer.training.extensions.snapshot()*

Methods

__call__ (*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

create_worker (*filename*, *outdir*, *target*, ***kwds*)
Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is already formated string.
- **outdir** (*str*) – Output directory. Passed by *trainer.out*.
- **target** (*dict*) – Serialized object which will be saved.
- **kwds** – Keyword arguments for the `savefun`.

finalize()
Finalizes the wirter.

Like extensions in `Trainer`, this method is invoked at the end of the training.

save(*filename*, *outdir*, *target*, *savefun*, ***kwds*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.7.5 chainer.training.extensions.snapshot_writers.QueueWriter

class `chainer.training.extensions.snapshot_writers.`**QueueWriter**(*savefun=<function save_npz>*, *task=None*)

Base class of queue snapshot writers.

This class is a base class of snapshot writers that use a queue. A Queue is created when this class is constructed, and every time when `__call__` is invoked, a snapshot task is put into the queue.

Parameters

- **savefun** – Callable object which is passed to the `create_task()` if the task is `None`. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **task** – Callable object. Its `__call__` must have a same interface to `Writer.__call__`. This object is directly put into the queue.

See also:

- `chainer.training.extensions.snapshot()`

Methods

__call__(*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

consume(*q*)

create_consumer(*q*)

create_queue()

create_task(*savefun*)

finalize()
Finalizes the wirter.

Like extensions in `Trainer`, this method is invoked at the end of the training.

save(*filename*, *outdir*, *target*, *savefun*, ***kwds*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.7.6 chainer.training.extensions.snapshot_writers.ThreadQueueWriter

class chainer.training.extensions.snapshot_writers.**ThreadQueueWriter**(*savefun=<function save_npz>*, *task=None*)

Snapshot writer that uses a thread queue.

This class creates a thread and a queue by `threading` and `queue` modules respectively. The thread will be a consumer of the queue, and the main thread will be a producer of the queue.

See also:

- `chainer.training.extensions.snapshot()`

Methods

__call__(*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

consume (*q*)

create_consumer (*q*)

create_queue ()

create_task (*savefun*)

finalize ()

Finalizes the wirter.

Like extensions in *Trainer*, this method is invoked at the end of the training.

save (*filename*, *outdir*, *target*, *savefun*, ***kwds*)

__eq__ ()

Return self==value.

__ne__ ()

Return self!=value.

__lt__ ()

Return self<value.

__le__ ()

Return self<=value.

__gt__ ()

Return self>value.

__ge__ ()

Return self>=value.

4.7.7 chainer.training.extensions.snapshot_writers.ProcessQueueWriter

class chainer.training.extensions.snapshot_writers.**ProcessQueueWriter** (*savefun=<function save_npz>*, *task=None*)

Snapshot writer that uses process queue.

This class creates a process and a queue by `multiprocessing` module. The process will be a consumer of this queue, and the main process will be a producer of this queue.

Note: Forking a new process from MPI process might be danger. Consider using *ThreadQueueWriter* instead of `ProcessQueueWriter` if you are using MPI.

See also:

- *chainer.training.extensions.snapshot()*

Methods

__call__(*filename*, *outdir*, *target*)
Invokes the actual snapshot function.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **outdir** (*str*) – Output directory. Corresponds to `Trainer.out`.
- **target** (*dict*) – Serialized object which will be saved.

consume(*q*)

create_consumer(*q*)

create_queue()

create_task(*savefun*)

finalize()
Finalizes the wirter.

Like extensions in `Trainer`, this method is invoked at the end of the training.

save(*filename*, *outdir*, *target*, *savefun*, ***kwds*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.8 Training Tools

Chainer provides a standard implementation of the training loops under the `chainer.training` module. It is built on top of many other core features of Chainer, including Variable and Function, Link/Chain/ChainList, Optimizer, Dataset, and Reporter/Summary. Compared to the training loop abstraction of other machine learning tool kits, Chainer's training framework aims at maximal flexibility, while keeps the simplicity for the typical usages. Most components are pluggable, and users can overwrite the definition.

The core of the training loop abstraction is `Trainer`, which implements the training loop itself. The training loop consists of two parts: one is `Updater`, which actually updates the parameters to train, and the other is `Extension` for arbitrary functionalities other than the parameter update.

Updater and some extensions use `chainer.dataset` and `Iterator` to scan the datasets and load mini-batches. The trainer also uses `Reporter` to collect the observed values, and some extensions use `DictSummary` to accumulate them and computes the statistics.

You can find many examples for the usage of this training utilities from the official examples. You can also search the extension implementations from *Extensions*.

4.8.1 Trainer

`chainer.training.Trainer`	The standard training loop in Chainer.

chainer.training.Trainer

class chainer.training.**Trainer**(*updater*, *stop_trigger=None*, *out='result'*, *extensions=None*)

The standard training loop in Chainer.

Trainer is an implementation of a training loop. Users can invoke the training by calling the `run()` method.

Each iteration of the training loop proceeds as follows.

- Update of the parameters. It includes the mini-batch loading, forward and backward computations, and an execution of the update formula. These are all done by the update object held by the trainer.
- Invocation of trainer extensions in the descending order of their priorities. A trigger object is attached to each extension, and it decides at each iteration whether the extension should be executed. Trigger objects are callable objects that take the trainer object as the argument and return a boolean value indicating whether the extension should be called or not.

Extensions are callable objects that take the trainer object as the argument. There are three ways to define custom extensions: inheriting the `Extension` class, decorating functions by `make_extension()`, and defining any callable including lambda functions. See `Extension` for more details on custom extensions and how to configure them.

Users can register extensions to the trainer by calling the `extend()` method, where some configurations can be added.

- Trigger object, which is also explained above. In most cases, `IntervalTrigger` is used, in which case users can simply specify a tuple of the interval length and its unit, like `(1000, 'iteration')` or `(1, 'epoch')`.
- The order of execution of extensions is determined by their priorities. Extensions of higher priorities are invoked earlier. There are three standard values for the priorities:
  - `PRIORITY_WRITER`. This is the priority for extensions that write some records to the `observation` dictionary. It includes cases that the extension directly adds values to the observation dictionary, or the extension uses the `chainer.report()` function to report values to the observation dictionary.
  - `PRIORITY_EDITOR`. This is the priority for extensions that edit the `observation` dictionary based on already reported values.
  - `PRIORITY_READER`. This is the priority for extensions that only read records from the `observation` dictionary. This is also suitable for extensions that do not use the `observation` dictionary at all.

The current state of the trainer object and objects handled by the trainer can be serialized through the standard serialization protocol of Chainer. It enables us to easily suspend and resume the training loop.

```
>>> serializers.save_npz('my.trainer', trainer)  # To suspend and save
>>> serializers.load_npz('my.trainer', trainer)  # To load and resume
```

The `snapshot()` method makes regular snapshots of the `Trainer` object during training.

Note: The serialization does not recover everything of the training loop. It only recovers the states which change over the training (e.g. parameters, optimizer states, the batch iterator state, extension states, etc.). You must initialize the objects correctly before deserializing the states.

On the other hand, it means that users can change the settings on deserialization. For example, the exit condition can be changed on the deserialization, so users can train the model for some iterations, suspend it, and then resume it with larger number of total iterations.

During the training, it also creates a `Reporter` object to store observed values on each update. For each iteration, it creates a fresh observation dictionary and stores it in the `observation` attribute.

Links of the target model of each optimizer are registered to the reporter object as observers, where the name of each observer is constructed as the format `<optimizer name><link name>`. The link name is given by the `chainer.Link.namedlink()` method, which represents the path to each link in the hierarchy. Other observers can be registered by accessing the reporter object via the `reporter` attribute.

The default trainer is *plain*, i.e., it does not contain any extensions.

Parameters

- **updater** (`Updater`) – Updater object. It defines how to update the models.
- **stop_trigger** – Trigger that determines when to stop the training loop. If it is not callable, it is passed to `IntervalTrigger`.
- **out** – Output directory.
- **extensions** – Extensions registered to the trainer.

Variables

- **updater** – The updater object for this trainer.
- **stop_trigger** – Trigger that determines when to stop the training loop. The training loop stops at the iteration on which this trigger returns `True`.
- **observation** – Observation of values made at the last update. See the `Reporter` class for details.
- **out** – Output directory.
- **reporter** – Reporter object to report observed values.

Methods

extend(*extension*, *name=None*, *trigger=None*, *priority=None*, *call_before_training=False*, ***kwargs*)

Registers an extension to the trainer.

`Extension` is a callable object which is called after each update unless the corresponding trigger object decides to skip the iteration. The order of execution is determined by priorities: extensions with higher priorities are called earlier in each iteration. Extensions with the same priority are invoked in the order of registrations.

If two or more extensions with the same name are registered, suffixes are added to the names of the second to last extensions. The suffix is `_N` where N is the ordinal of the extensions.

See `Extension` for the interface of extensions.

Parameters

- **extension** – Extension to register.
- **name** (*str*) – Name of the extension. If it is omitted, the `Extension.name` attribute of the extension is used or the `Extension.default_name` attribute of the extension if *name* is is set to *None* or is undefined. Note that the name would be suffixed by an ordinal in case of duplicated names as explained above.
- **trigger** (*tuple or Trigger*) – Trigger object that determines when to invoke the extension. If it is `None`, `extension.trigger` is used instead. If it is `None` and the extension does not have the trigger attribute, the extension is triggered at every iteration by default. If the trigger is not callable, it is passed to `IntervalTrigger` to build an interval trigger.
- **call_before_training** (*bool*) – Flag to call extension before training. Default is `False`.
- **priority** (*int*) – Invocation priority of the extension. Extensions are invoked in the descending order of priorities in each iteration. If this is `None`, `extension.priority` is used instead.

get_extension(*name*)
Returns the extension of a given name.

Parameters **name** (*str*) – Name of the extension.

Returns Extension.

run(*show_loop_exception_msg=True*)
Executes the training loop.

This method is the core of `Trainer`. It executes the whole loop of training the models.

Note that this method cannot run multiple times for one trainer object.

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

elapsed_time
Total time used for the training.

The time is in seconds. If the training is resumed from snapshot, it includes the time of all the previous training to get the current state of the trainer.

is_before_training
Flag that represents if training has started or not.

`True` represents 'before training' and `False` represents 'during/after training'.

This flag is supposed to be used in *Extension.__call__()* (e.g., `PlotReport.__call__()`) to decide to execute its operation or not. This additional condition is necessary since `Extension._trigger(trainer)` is always `False` before training and cannot be used.

4.8.2 Updaters

chainer.training.Updater	Interface of updater objects for trainers.
chainer.training.updaters.StandardUpdater	Standard implementation of Updater.
chainer.training.updaters.ParallelUpdater	Implementation of a parallel GPU Updater.
chainer.training.updaters.MultiprocessParallelUpdater	Implementation of a multiprocess parallel GPU Updater.

chainer.training.Updater

class chainer.training.**Updater**
Interface of updater objects for trainers.

Updater implements a training iteration as *update()*. Typically, the updating iteration proceeds as follows.

- Fetch a minibatch from *dataset* via *Iterator*.
- Run forward and backward process of *Chain*.
- Update parameters according to their *UpdateRule*.

The first line is processed by *Iterator.__next__*. The second and third are processed by *Optimizer.update*. Users can also implement their original updating iteration by overriding *Updater.update*.

Methods

connect_trainer(*trainer*)
Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of *Trainer*. The default implementation does nothing.

Parameters trainer (`Trainer`) – Trainer object to which the updater is registered.

finalize()
Finalizes the updater object.

This method is called at the end of training loops. It should finalize each dataset iterator used in this updater.

get_all_optimizers()
Gets a dictionary of all optimizers for this updater.

Returns Dictionary that maps names to optimizers.

Return type dict

get_optimizer(*name*)
Gets the optimizer of given name.

Updater holds one or more optimizers with names. They can be retrieved by this method.

Parameters **name** (*str*) – Name of the optimizer.

Returns Optimizer of the name.

Return type *Optimizer*

serialize(*serializer*)
Serializes the current state of the updater object.

update()
Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.

This method is called once at each iteration of the training loop.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.training.updaters.StandardUpdater

class chainer.training.updaters.**StandardUpdater**(*iterator*, *optimizer*, *converter=convert.concat_examples*, *device=None*, *loss_func=None*, *loss_scale=None*, *auto_new_epoch=True*, ***, *input_device=None*)
Standard implementation of Updater.

This is the standard implementation of `Updater`. It accepts one or more training datasets and one or more optimizers. The default update routine assumes that there is only one training dataset and one optimizer. Users can override this update routine by inheriting this class and overriding the `update_core()` method. Each batch is converted to input arrays by `chainer.dataset.concat_examples()` by default, which can also be manually set by `converter` argument.

Parameters

- **iterator** – Dataset iterator for the training dataset. It can also be a dictionary that maps strings to iterators. If this is just an iterator, then the iterator is registered by the name `'main'`.
- **optimizer** – Optimizer to update parameters. It can also be a dictionary that maps strings to optimizers. If this is just an optimizer, then the optimizer is registered by the name `'main'`.
- **converter** – Converter function to build input arrays. Each batch extracted by the main iterator and the `device` option are passed to this function. `chainer.dataset.concat_examples()` is used by default.
- **device** (*device specifier*) – Device to which the model is sent. If `None`, the device of the model will stay unchanged.
- **loss_func** – Loss function. The target link of the main optimizer is used by default.
- **loss_scale** (*float*) – Loss scaling factor. Loss scaling is a usefull technique to mitigate vanishing gradient issue that tends to happen when low precision data type like float16 is used during training. If you set loss scaling factor, gradients of loss values are to be multiplied by the factor before backprop starts. The factor is propagated to whole gradients in a computational graph along the backprop. The gradients of parameters are divided by the factor just before the parameters are to be updated.
- **auto_new_epoch** (*bool*) – If `True`, `new_epoch()` of the main optimizer is automatically called when the `is_new_epoch` attribute of the main iterator is `True`.
- **input_device** (*device specifier*) – Device to which the training data is sent. If `input_device` is omitted, it will match the `device` argument.

Variables

- **converter** – Converter function.
- **loss_func** – Loss function. If it is `None`, the target link of the main optimizer is used instead.
- ***device*** – Device to which the model is sent.
- ***input_device*** – Device to which the training data is sent.
- **iteration** – Current number of completed updates.
- **auto_new_epoch** – If `True`, `new_epoch()` is automatically called by `update_core()`. In this case, the `use_auto_new_epoch` attribute of each optimizer is also set to `True`. If `update_core()` is overridden, the implementation should correctly call `new_epoch()` of each optimizer.

Methods

connect_trainer(*trainer*)

Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of `Trainer`. The default implementation does nothing.

Parameters **trainer** (`Trainer`) – Trainer object to which the updater is registered.

finalize()

Finalizes the updater object.

This method calls the *finalize* method of each iterator that this updater has. It is called at the end of training loops.

get_all_optimizers()
Gets a dictionary of all optimizers for this updater.

Returns Dictionary that maps names to optimizers.

Return type dict

get_iterator(*name*)
Gets the dataset iterator of given name.

Parameters **name** (*str*) – Name of the dataset iterator.

Returns Corresponding dataset iterator.

Return type *Iterator*

get_optimizer(*name*)
Gets the optimizer of given name.

Parameters **name** (*str*) – Name of the optimizer.

Returns Corresponding optimizer.

Return type *Optimizer*

serialize(*serializer*)
Serializes the current state of the updater object.

update()
Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.

This method is called once at each iteration of the training loop.

update_core()

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device

epoch

epoch_detail

input_device

is_new_epoch

previous_epoch_detail

chainer.training.updaters.ParallelUpdater

class chainer.training.updaters.**ParallelUpdater**(*iterator*, *optimizer*, *converter=<chainer.dataset.convert._ArbitraryCallableConverter object>*, *models=None*, *devices=None*, *loss_func=None*, *loss_scale=None*, *auto_new_epoch=True*)

Implementation of a parallel GPU Updater.

This is an implementation of `Updater` that uses multiple GPUs. It behaves similarly to *StandardUpdater*. The update routine is modified to support data-parallel computation on multiple GPUs in one machine. It is based on synchronous parallel SGD: it parallelizes the gradient computation over a mini-batch, and updates the parameters only in the main device.

Parameters

- **iterator** – Dataset iterator for the training dataset. It can also be a dictionary that maps strings to iterators. If this is just an iterator, then the iterator is registered by the name `'main'`.
- **optimizer** – Optimizer to update parameters. It can also be a dictionary that maps strings to optimizers. If this is just an optimizer, then the optimizer is registered by the name `'main'`.
- **converter** – Converter function to build input arrays. Each batch extracted by the main iterator is split equally between the devices and then passed with corresponding `device` option to this function. *concat_examples()* is used by default.
- **models** – Dictionary of models. The main model should be the same model attached to the `'main'` optimizer.
- **devices** – Dictionary of devices to which the training data is sent. The devices should be arranged in a dictionary with the same structure as `models`.
- **loss_func** – Loss function. The model is used as a loss function by default.
- **loss_scale** (*float*) – Loss scaling factor. Loss scaling is a usefull technique to mitigate vanishing gradient issue that tends to happen when low precision data type like float16 is used during training. If you set loss scaling factor, gradients of loss values are to be multiplied by the factor before backprop starts. The factor is propagated to whole gradients in a computational graph along the backprop. The gradients of parameters are divided by the factor just before the parameters are to be updated.
- **auto_new_epoch** (*bool*) – If `True`, *new_epoch()* of the main optimizer is automatically called when the `is_new_epoch` attribute of the main iterator is `True`.

Methods

connect_trainer(*trainer*)
Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of `Trainer`. The default implementation does nothing.

Parameters **trainer** (`Trainer`) – Trainer object to which the updater is registered.

finalize()
Finalizes the updater object.

This method calls the *finalize* method of each iterator that this updater has. It is called at the end of training loops.

get_all_optimizers()
Gets a dictionary of all optimizers for this updater.

Returns Dictionary that maps names to optimizers.

Return type dict

get_iterator(*name*)
Gets the dataset iterator of given name.

Parameters **name** (*str*) – Name of the dataset iterator.

Returns Corresponding dataset iterator.

Return type *Iterator*

get_optimizer(*name*)
Gets the optimizer of given name.

Parameters **name** (*str*) – Name of the optimizer.

Returns Corresponding optimizer.

Return type *Optimizer*

serialize(*serializer*)
Serializes the current state of the updater object.

update()
Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.

This method is called once at each iteration of the training loop.

update_core()

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device

epoch

epoch_detail

input_device

is_new_epoch

previous_epoch_detail

chainer.training.updaters.MultiprocessParallelUpdater

class chainer.training.updaters.**MultiprocessParallelUpdater**(*iterators*, *optimizer*, *converter=<chainer.dataset.convert._ArbitraryC object>*, *devices=None*, *auto_new_epoch=True*)

Implementation of a multiprocess parallel GPU Updater.

This is an implementation of `Updater` that uses multiple GPUs with multi-process data parallelism. It uses Nvidia NCCL for communication between multiple GPUs.

It behaves similarly to *StandardUpdater*. The update routine is modified to support data-parallel computation on multiple GPUs in one machine. It is based on synchronous parallel SGD: it parallelizes the gradient computation over a mini-batch, and updates the parameters only in the main device.

It does not transfer the values collected by `Reporter` in the sub devices to the main device. So you can only see the reported values in the main device.

Parameters

- **iterators** – List of dataset iterator for the training dataset. The number of the iterators must be same to the number of GPUs you use.
- **optimizer** – Optimizer to update parameters. The model should be attached to the optimizer.
- **converter** – Converter function to build input arrays. Each batch extracted by the iterator is split equally between the devices and then passed with corresponding `device` option to this function. *concat_examples()* is used by default.
- **devices** – Dictionary or list of devices to which the training data is sent. The master device will be the first one in the list or the value attached to the key `'main'`.
- **auto_new_epoch** (*bool*) – If `True`, *new_epoch()* of the main optimizer is automatically called when the `is_new_epoch` attribute of the main iterator is `True`.

Methods

static available()

connect_trainer(*trainer*)

Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of *Trainer*. The default implementation does nothing.

Parameters **trainer** (Trainer) – Trainer object to which the updater is registered.

finalize()

Finalizes the updater object.

This method calls the *finalize* method of each iterator that this updater has. It is called at the end of training loops.

get_all_optimizers()

Gets a dictionary of all optimizers for this updater.

Returns Dictionary that maps names to optimizers.

Return type dict

get_iterator(*name*)

Gets the dataset iterator of given name.

Parameters **name** (*str*) – Name of the dataset iterator.

Returns Corresponding dataset iterator.

Return type *Iterator*

get_optimizer(*name*)

Gets the optimizer of given name.

Parameters **name** (*str*) – Name of the optimizer.

Returns Corresponding optimizer.

Return type *Optimizer*

serialize(*serializer*)

Serializes the current state of the updater object.

setup_workers()

update()

Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.

This method is called once at each iteration of the training loop.

update_core()

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

device

epoch

epoch_detail

input_device

is_new_epoch

previous_epoch_detail

We have two kinds of updaters for multi-gpus training. The pros/cons for the updaters are as follows:

ParallelUpdater:

- (+) Can use the same iterator for any number of GPUs
- (-) No parallelism at CPU side
- (-) GPUs used later may be blocked due to the limit of kernel-launch queue size

MultiprocessParallelUpdater:

- (+) Parallelism at CPU side
- (+) No degrade due to kernel launch queue size
- (-) Need per-process data iterator
- (-) Reporter cannot collect data except for one of the devices

4.8.3 Extensions

An extension is a callable object that can perform arbitrary actions during the training loop. Extensions can be registered to `Trainer` by using `Trainer.extend()` method, and they are invoked when the *Trigger* condition is satisfied.

In addition to the built-in extensions listed below, you can define your own extension by implementing `Extension` or using the `make_extension()` decorator. See *Trainer Extensions* for details.

Common

`chainer.training.Extension`	Base class of trainer extensions.
`chainer.training.make_extension`	Decorator to make given functions into trainer extensions.

chainer.training.Extension

class chainer.training.**Extension**

Base class of trainer extensions.

Extension of `Trainer` is a callable object that takes the trainer object as the argument. It also provides some default configurations as its attributes, e.g. the default trigger and the default priority. This class provides a set of typical default values for these attributes.

There are three ways to define users' own extensions: inheriting this class, decorating closures by `make_extension()`, or using any callable including lambda functions as extensions. Decorator can slightly reduce the overhead and is much easier to use, while this class provides more flexibility (for example, it can have methods to configure the behavior). Using a lambda function allows one-line coding for simple purposes, but users have to specify the configurations as arguments to *Trainer.extend()*. For a callable not inheriting this class, the default configurations of this class are used unless the user explicitly specifies them in *Trainer.extend()* method.

Variables

- ***trigger*** – Default value of trigger for this extension. It is set to `(1, 'iteration')` by default.
- ***priority*** – Default priority of the extension. It is set to `PRIORITY_READER` by default.
- ***name*** – Name of the extension. It is set to `None` by default. This value will be overwritten when registering an extension to a trainer. See *chainer.training.Trainer.extend()* for details.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (Trainer) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.make_extension

chainer.training.**make_extension**(*trigger=None*, *default_name=None*, *priority=None*, *finalizer=None*, *initializer=None*, *on_error=None*, ***kwargs*)
Decorator to make given functions into trainer extensions.

This decorator just adds some attributes to a given function. The value of the attributes are given by the arguments of this decorator.

See `Extension` for details of trainer extensions. Most of the default values of arguments also follow those for this class.

Parameters

- **trigger** – Default trigger of the extension.
- **default_name** – Default name of the extension. The name of a given function is used by default.
- **priority** (*int*) – Default priority of the extension.
- **finalizer** – Finalizer function of this extension. It is called at the end of the training loop.

- **initializer** – Initializer function of this extension. It is called at the beginning of the training loop.
- **on_error** – Error handler callback function of this extension. It is called after an error is raised during the trainer loop.

Evaluation and Metrics Collection

These extensions provide features to collect additional metrics. The typical use case is to use *Evaluator* to perform evaluation with a validation dataset to compute validation loss/accuracy.

chainer.training.extensions.Evaluator	Trainer extension to evaluate models on a validation set.
chainer.training.extensions.MicroAverage	Calculates micro-average ratio.
chainer.training.extensions.FailOnNonNumber	Trainer extension to raise RuntimeError if parameters contain NaN or Inf.
chainer.training.extensions.ParameterStatistics	Trainer extension to report parameter statistics.
chainer.training.extensions.observe_lr	Returns a trainer extension to record the learning rate.
chainer.training.extensions.observe_value	Returns a trainer extension to continuously record a value.

chainer.training.extensions.Evaluator

class chainer.training.extensions.**Evaluator**(*self*, *iterator*, *target*, *converter=convert.concat_examples*, *device=None*, *eval_hook=None*, *eval_func=None*, ***, *progress_bar=False*)

Trainer extension to evaluate models on a validation set.

This extension evaluates the current models by a given evaluation function. It creates a *Reporter* object to store values observed in the evaluation function on each iteration. The report for all iterations are aggregated to *DictSummary*. The collected mean values are further reported to the reporter object of the trainer, where the name of each observation is prefixed by the evaluator name. See *Reporter* for details in naming rules of the reports.

Evaluator has a structure to customize similar to that of *StandardUpdater*. The main differences are:

- There are no optimizers in an evaluator. Instead, it holds links to evaluate.
- An evaluation loop function is used instead of an update function.
- Preparation routine can be customized, which is called before each evaluation. It can be used, e.g., to initialize the state of stateful recurrent networks.

There are two ways to modify the evaluation behavior besides setting a custom evaluation function. One is by setting a custom evaluation loop via the `eval_func` argument. The other is by inheriting this class and overriding the *evaluate()* method. In latter case, users have to create and handle a reporter object manually. Users also have to copy the iterators before using them, in order to reuse them at the next time of evaluation. In both cases, the functions are called in testing mode (i.e., `chainer.config.train` is set to `False`).

This extension is called at the end of each epoch by default.

Parameters

- **iterator** – Dataset iterator for the validation dataset. It can also be a dictionary of iterators. If this is just an iterator, the iterator is registered by the name `'main'`.
- **target** – Link object or a dictionary of links to evaluate. If this is just a link object, the link is registered by the name `'main'`.
- **converter** – Converter function to build input arrays. *concat_examples()* is used by default.
- **device** – Device to which the validation data is sent. Negative value indicates the host memory (CPU).
- **eval_hook** – Function to prepare for each evaluation process. It is called at the beginning of the evaluation. The evaluator extension object is passed at each call.
- **eval_func** – Evaluation function called at each iteration. The target link to evaluate as a callable is used by default.
- **progress_bar** – Boolean flag to show a progress bar while training, which is similar to *ProgressBar*. (default: `False`)

Warning: The argument `progress_bar` is experimental. The interface can change in the future.

Variables

- **converter** – Converter function.
- **device** – Device to which the validation data is sent.
- **eval_hook** – Function to prepare for each evaluation process.
- **eval_func** – Evaluation function called at each iteration.

Methods

__call__(*trainer=None*)

Executes the evaluator extension.

Unlike usual extensions, this extension can be executed without passing a trainer object. This extension reports the performance on validation dataset using the *report()* function. Thus, users can use this extension independently from any trainer by manually configuring a *Reporter* object.

Parameters **trainer** (Trainer) – Trainer object that invokes this extension. It can be omitted in case of calling this extension manually.

Returns Result dictionary that contains mean statistics of values reported by the evaluation function.

Return type dict

evaluate()

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to *DictSummary* and returns a dictionary whose values are means computed by the summary.

Note that this function assumes that the main iterator raises `StopIteration` or code in the evaluation loop raises an exception. So, if this assumption is not held, the function could be caught in an infinite loop.

Users can override this method to customize the evaluation routine.

Note: This method encloses `eval_func` calls with `function.no_backprop_mode()` context, so all calculations using *FunctionNode*s inside `eval_func` do not make computational graphs. It is for reducing the memory consumption.

Returns Result dictionary. This dictionary is further reported via *report()* without specifying any observer.

Return type dict

finalize()
Finalizes the evaluator object.

This method calls the *finalize* method of each iterator that this evaluator has. It is called at the end of training loops.

get_all_iterators()
Returns a dictionary of all iterators.

get_all_targets()
Returns a dictionary of all target links.

get_iterator(*name*)
Returns the iterator of the given name.

get_target(*name*)
Returns the target link of the given name.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name = 'validation'

name = None

priority = 300

trigger = (1, 'epoch')

chainer.training.extensions.MicroAverage

class chainer.training.extensions.**MicroAverage**(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

Calculates micro-average ratio.

Give N batches and values $\{n_1, \dots, n_N\}$ and $\{d_1, \dots, d_N\}$, this extension calculates micro-average of these ratio defined as:

$$\frac{\sum_i^N n_i}{\sum_i^N d_i}.$$

A user usually uses the number of examples which a system correctly predict as n_i and the number of total examples in i-th batch as d_i. This value is called macro-average of precision.

Note that macro-average is defined as:

$$\frac{1}{N}\sum_i^N (n_i/d_i),$$

It is same to the micro-average when each mini-batch has the same d_i.

You need to report numerator value (the number of correct examples) and denominator value (the number of examples) in your model.

```
>>> class MyModel(chainer.Link):
...     def __call__(self, x, y):
...         loss = F.softmax_cross_entropy(x, y)
...         correct = (x.data.argmax(axis=1) == y.data).sum()
...         total = len(y.data)
...         reporter.report({'correct': correct, 'total': total}, self)
...         return loss
```

And then, make an extension with corresponding reporting keys and register it.

```
>>> ext = extensions.MicroAverage(
...     'main/correct', 'main/total', 'main/accuracy')
```

Parameters

- **numerator_key** (*str*) – Key string of obserbation storing a numerator value.
- **denominator_key** (*str*) – Key string of obserbation storing a denominator value.
- **result_key** (*str*) – Key string of obserbation to store a result.
- **trigger** – Trigger that decides when to calcurate average. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 200

trigger = (1, 'iteration')

chainer.training.extensions.FailOnNonNumber

class `chainer.training.extensions.`**FailOnNonNumber**
Trainer extension to raise RuntimeError if parameters contain NaN or Inf.

Although parameters including non-number such as NaN and Inf are unnecessary in most cases, *`Trainer`* will continue to compute even if the parameters in a given optimizer diverge. This extension is aimed to reduce unnecessary computations by throwing `RuntimeError` if the parameters contain NaN or Inf.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.ParameterStatistics

class chainer.training.extensions.**ParameterStatistics**(*links*, *statistics='default'*, *report_params=True*, *report_grads=True*, *prefix=None*, *trigger=(1, 'epoch')*, *skip_nan_params=False*)

Trainer extension to report parameter statistics.

Statistics are collected and reported for a given `Link` or an iterable of `Link`s. If a link contains child links, the statistics are reported separately for each child.

Any function that takes a one-dimensional `numpy.ndarray` or a `cupy.ndarray` and outputs a single or multiple real numbers can be registered to handle the collection of statistics, e.g. `numpy.ndarray.mean()`.

The keys of reported statistics follow the convention of link name followed by parameter name, attribute name and function name, e.g. `VGG16Layers/conv1_1/W/data/mean`. They are prepended with an optional prefix and appended with integer indices if the statistics generating function return multiple values.

Parameters

- **links** (`Link or iterable of ~chainer.Link`) – Link(s) containing the parameters to observe. The link is expected to have a `name` attribute which is used as a part of the report key.
- **statistics** (*dict or 'default'*) – Dictionary with function name to function mappings. The name is a string and is used as a part of the report key. The function is responsible for generating the statistics. If the special value `'default'` is specified, the default statistics functions will be used.
- **report_params** (*bool*) – If `True`, report statistics for parameter values such as weights and biases.
- **report_grads** (*bool*) – If `True`, report statistics for parameter gradients.
- **prefix** (*str*) – Optional prefix to prepend to the report keys.
- **trigger** – Trigger that decides when to aggregate the results and report the values.
- **skip_nan_params** (*bool*) – If `True`, statistics are not computed for parameters including NaNs and a single NaN value is immediately reported instead. Otherwise, this extension will simply try to compute the statistics without performing any checks for NaNs.

Note: The default statistic functions are as follows:

- `'mean'` (`xp.mean(x)`)
- `'std'` (`xp.std(x)`)
- `'min'` (`xp.min(x)`)
- `'max'` (`xp.max(x)`)
- `'zeros'` (`xp.count_nonzero(x == 0)`)
- `'percentile'` (`xp.percentile(x, (0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87))`)

Methods

__call__(*trainer*)
Execute the statistics extension.

Collect statistics for the current state of parameters.

Note that this method will merely update its statistic summary, unless the internal trigger is fired. If the trigger is fired, the summary will also be reported and then reset for the next accumulation.

Parameters **trainer** (Trainer) – Associated trainer that invoked this extension.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

register_statistics(*name*, *function*)
Register a function to compute a certain statistic.

The registered function will be called each time the extension runs and the results will be included in the report.

Parameters

- **name** (*str*) – Name of the statistic.
- **function** – Function to generate the statistic. Any function that takes a one-dimensional `numpy.ndarray` or a `cupy.ndarray` and outputs a single or multiple real numbers is allowed.

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name = 'parameter_statistics'

default_statistics = {'max': <function <lambda>>, 'mean': <function <lambda>>, 'min'

name = None

priority = 300

report_key_template = '{prefix}{link_name}{param_name}/{attr_name}/{function_name}'

trigger = (1, 'iteration')

chainer.training.extensions.observe_lr

`chainer.training.extensions.`**observe_lr**(*optimizer_name='main'*, *observation_key='lr'*)
Returns a trainer extension to record the learning rate.

Parameters

- **optimizer_name** (*str*) – Name of optimizer whose learning rate is recorded.
- **observation_key** (*str*) – Key of observation to record.

Returns The extension function.

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the *Trainer.extend()* method.

chainer.training.extensions.observe_value

`chainer.training.extensions.`**observe_value**(*observation_key*, *target_func*)
Returns a trainer extension to continuously record a value.

Parameters

- **observation_key** (*str*) – Key of observation to record.
- **target_func** (*function*) – Function that returns the value to record. It must take one argument: :class:~chainer.training.Trainer object.

Returns The extension function.

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the *Trainer.extend()* method.

Optimizer Behavior Control

These extensions provide features to adjust optimizer behavior. The typical use case is to change the learning rate of the optimizer over time.

chainer.training.extensions.ExponentialShift	Trainer extension to exponentially shift an optimizer attribute.
chainer.training.extensions.InverseShift	Trainer extension to shift an optimizer attribute.
chainer.training.extensions.LinearShift	Trainer extension to change an optimizer attribute linearly.
chainer.training.extensions.MultistepShift	Trainer extension to shift an optimizer attribute in several steps.
chainer.training.extensions.PolynomialShift	Trainer extension to polynomially shift an optimizer attribute.
chainer.training.extensions.WarmupShift	Trainer extension to gradually initialize an optimizer attribute.
chainer.training.extensions.StepShift	Trainer extension to shift an optimizer attribute in "steps".

chainer.training.extensions.ExponentialShift

class chainer.training.extensions.**ExponentialShift**(*attr*, *rate*, *init=None*, *target=None*, *optimizer=None*)

Trainer extension to exponentially shift an optimizer attribute.

This extension exponentially increases or decreases the specified attribute of the optimizer. The typical use case is an exponential decay of the learning rate.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the attribute to shift.
- **rate** (*float*) – Rate of the exponential shift. This value is multiplied to the attribute at each call.
- **init** (*float*) – Initial value of the attribute. If it is `None`, the extension extracts the attribute at the first call and uses it as the initial value.
- **target** (*float*) – Target value of the attribute. If the attribute reaches this value, the shift stops.
- **optimizer** (`Optimizer`) – Target optimizer to adjust the attribute. If it is `None`, the main optimizer of the updater is used.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *`Trainer`* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *`ExponentialShift`* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *`ExponentialShift`* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*`Exception`*) – arbitrary exception thrown during update loop.
- **tb** (*`traceback`*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.InverseShift

class chainer.training.extensions.**InverseShift**(*attr*, *gamma*, *power*, *init=None*, *target=None*, *optimizer=None*)

Trainer extension to shift an optimizer attribute.

The new value is computed according to the fomula below: new_attr = init_attr * (1 + gamma * iter) ^ (- power), which is compatible to the `inv` learning rate policy in Caffe.

The typical use is to decrease the learning rate during the training.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the attribute to shift.
- **gamma** (*float*) – Parameter used to compute the new value. Refer to the fomula above. Note that gamma is assumed to be nonegative.
- **power** (*float*) – Parameter used to compute the new value. Refer to the fomula above.
- **init** (*float*) – Initial value of the attribute. If it is `None`, the extension extracts the attribute at the first call and uses it as the initial value.
- **target** (*float*) – Target value of the attribute. If the attribute reaches this value, the shift stops.
- **optimizer** (`Optimizer`) – Target optimizer to adjust the attribute. If it is `None`, the main optimizer of the updater is used.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.LinearShift

class chainer.training.extensions.**LinearShift**(*attr*, *value_range*, *time_range*, *optimizer=None*)

Trainer extension to change an optimizer attribute linearly.

This extension changes an optimizer attribute from the first value to the last value linearly within a specified duration. The typical use case is warming up of the momentum coefficient.

For example, suppose that this extension is called at every iteration, and `value_range == (x, y)` and `time_range == (i, j)`. Then, this extension keeps the attribute to be `x` up to the `i`-th iteration, linearly shifts the value to `y` by the `j`-th iteration, and then keeps the value to be `y` after the `j`-th iteration.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the optimizer attribute to adjust.
- **value_range** (*tuple of float*) – The first and the last values of the attribute.
- **time_range** (*tuple of ints*) – The first and last counts of calls in which the attribute is adjusted.
- **optimizer** (`Optimizer`) – Target optimizer object. If it is None, the main optimizer of the trainer is used.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*`Exception`*) – arbitrary exception thrown during update loop.
- **tb** (*`traceback`*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.MultistepShift

class chainer.training.extensions.**MultistepShift**(*attr*, *gamma*, *step_value*, *init*, *optimizer=None*)
Trainer extension to shift an optimizer attribute in several steps.

This extension changes an optimizer attribute in several steps, every step the attribute will multiply a factor `gamma`.

For example, suppose that this extension is called at every iteration, and `init = x`, `gamma = y`, `step_value = [s1, s2, s3]`. Then during the iterations from 0 to (s1 - 1), the attr will be `x`. During the iterations from s1 to (s2 - 1), the attr will be `x * y`. During the iterations from s2 to (s3 - 1), the attr will be `x * y * y`. During the iterations after s3, the attr will be `x * y * y * y`.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the attribute to shift.
- **init** (*float*) – Initial value of the attribute. If it is `None`, the extension extracts the attribute at the first call and uses it as the initial value.
- **gamma** (*float*) – The factor which the attr will mutiply at the beginning of each step.
- **step_value** (*tuple*) – The first iterations of each step.
- **optimizer** (`Optimizer`) – Target optimizer to adjust the attribute. If it is `None`, the main optimizer of the updater is used.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.

- **tb** (*traceback*) – traceback object of the exception

serialize (*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.PolynomialShift

class chainer.training.extensions.**PolynomialShift** (*attr*, *rate*, *max_count*, *init=None*, *target=None*, *optimizer=None*)

Trainer extension to polynomially shift an optimizer attribute.

This extension polynomially decreases the specified attribute of the optimizer. The typical use case is a polynomial decay of the learning rate at each iteration.

For example, suppose that this extension is invoke at every iteration. Then this extension will set the corresponding attribute to `init_value * (1 - i / max_iter) ^ rate` at the `i`-th iteration, where the `max_iter` is the number of iterations to be running.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the attribute to shift.
- **rate** (*float*) – Exponent of polynomial shift.
- **max_count** (*int*) – Number of this extension to be invoked.

- **init** (*float*) – Initial value of the attribute. If it is None, the extension extracts the attribute at the first call and uses it as the initial value.
- **target** (*float*) – Target value of the attribute. If the attribute reaches this value, the shift stops.
- **optimizer** (Optimizer) – Target optimizer to adjust the attribute. If it is None, the main optimizer of the updater is used.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (Trainer) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

invoke_before_training = True

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.WarmupShift

class chainer.training.extensions.**WarmupShift**(*attr*, *warmup_start*, *warmup_iter*, *init*, *optimizer=None*)

Trainer extension to gradually initialize an optimizer attribute.

This extension changes an optimizer attribute evenly at the begining of one training.

For example, suppose that this extension is called at every iteration, and warmup_start = x , init = y, warmup_iter = t. Then this extension will set the corresponding attribute to from `x` to `y` evenly in first `t` iterations.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the optimizer attribute to adjust.
- **warmup_start** (*float*) – the value of the attr at the begining of one training.
- **init** (*float*) – the value of the attr after warm up iterations.
- **warmup_iter** (*int*) – the number of the iterations in which the attr changes from `warmup_start` to `init`.
- **optimizer** (`Optimizer`) – Target optimizer object. If it is None, the main optimizer of the trainer is used.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)

Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

default_name

Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.StepShift

class chainer.training.extensions.**StepShift**(*attr*, *gamma*, *step*, *init=None*, *target=None*, *optimizer=None*)

Trainer extension to shift an optimizer attribute in "steps".

This extension multiplies the specified attribute of the optimizer in "steps". The typical use case is to scale the attribute at every `k`th iteration.

For example, suppose that this extension is invoked at every iteration, then given `k`, a multiplier `gamma` and an initial value `init`, the optimizer attribute is set to `init * gamma ^ (floor(i / k))`, where `i` represents the index of the current iteration.

This extension is also called before the training loop starts by default.

Parameters

- **attr** (*str*) – Name of the optimizer attribute to adjust.
- **gamma** (*float*) – The multiplier.
- **step** (*int*) – The interval for the multiplication, i.e., `k`.
- **init** (*float*) – Initial value of the attribute. If it is `None`, the extension extracts the attribute at the first call and uses it as the initial value.
- **target** (*float*) – Target value of the attribute. If the attribute reaches this value, the shift stops.
- **optimizer** (`Optimizer`) – Target optimizer object. If it is None, the main optimizer of the trainer is used.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

Reporting

These extensions provide features to perform reporting of metrics and various statistics to the console or files.

chainer.training.extensions.PrintReport	Trainer extension to print the accumulated results.
chainer.training.extensions.ProgressBar	Trainer extension to print a progress bar and recent training status.
chainer.training.extensions.LogReport	Trainer extension to output the accumulated results to a log file.
chainer.training.extensions.PlotReport	Trainer extension to output plots.
chainer.training.extensions.VariableStatisticsPlot	Trainer extension to plot statistics for *Variable*s.
chainer.training.extensions.DumpGraph	Trainer extension to dump a computational graph.

chainer.training.extensions.PrintReport

class chainer.training.extensions.**PrintReport**(*entries*, *log_report='LogReport'*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Trainer extension to print the accumulated results.

This extension uses the log accumulated by a *LogReport* extension to print specified entries of the log in a human-readable format.

Parameters

- **entries** (*list of str*) – List of keys of observations to print.
- **log_report** (*str or LogReport*) – Log report to accumulate the observations. This is either the name of a LogReport extensions registered to the trainer, or a LogReport instance to use internally.
- **out** – Stream to print the bar. Standard output is used by default.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (*Trainer*) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.ProgressBar

class chainer.training.extensions.**ProgressBar**(*training_length=None*, *update_interval=100*, *bar_length=50*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Trainer extension to print a progress bar and recent training status.

This extension prints a progress bar at every call. It watches the current iteration and epoch to print the bar.

Parameters

- **training_length** (*tuple*) – Length of whole training. It consists of an integer and either 'epoch' or 'iteration'. If this value is omitted and the stop trigger of the trainer is IntervalTrigger, this extension uses its attributes to determine the length of the training.
- **update_interval** (*int*) – Number of iterations to skip printing the progress bar.
- **bar_length** (*int*) – Length of the progress bar in characters.
- **out** – Stream to print the bar. Standard output is used by default.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (Trainer) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (Trainer) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.LogReport

class chainer.training.extensions.**LogReport**(*keys=None, trigger=(1, 'epoch'), postprocess=None, filename='log'*)
Trainer extension to output the accumulated results to a log file.

This extension accumulates the observations of the trainer to *DictSummary* at a regular interval specified by a supplied trigger, and writes them into a log file in JSON format.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds some entries to each result dictionary.

- `'epoch'` and `'iteration'` are the epoch and iteration counts at the output, respectively.

- `'elapsed_time'` is the elapsed time in seconds since the training begins. The value is taken from `Trainer.elapsed_time`.

Parameters

- **keys** (*iterable of strs*) – Keys of values to accumulate. If this is None, all the values are accumulated and output to the log file.
- **trigger** – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **postprocess** – Callback to postprocess the result dictionaries. Each result dictionary is passed to this callback on the output. This callback can modify the result dictionaries, which are used to output to the log file.
- **filename** (*str*) – Name of the log file under the output directory. It can be a format string: the last result dictionary is passed for the formatting. For example, users can use '{iteration}' to separate the log files for different iterations. If the log name is None, it does not output the log to any file. For historical reasons `log_name` is also accepted as an alias of this argument.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer, exc, tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.

- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize (*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

log
The current list of observation dictionaries.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.PlotReport

class chainer.training.extensions.**PlotReport**(*y_keys*, *x_key='iteration'*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename='plot.png'*, *marker='x'*, *grid=True*)

Trainer extension to output plots.

This extension accumulates the observations of the trainer to *DictSummary* at a regular interval specified by a supplied trigger, and plot a graph with using them.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds `'epoch'` and `'iteration'` entries to each result dictionary, which are the epoch and iteration counts at the output.

> **Warning:** If your environment needs to specify a backend of matplotlib explicitly, please call `matplotlib.use` before calling `trainer.run`. For example:
>
> ```
> import matplotlib
> matplotlib.use('Agg')
>
> trainer.extend(
>     extensions.PlotReport(['main/loss', 'validation/main/loss'],
>                           'epoch', filename='loss.png'))
> trainer.run()
> ```
>
> Then, once one of instances of this extension is called, `matplotlib.use` will have no effect.

For the details, please see here: https://matplotlib.org/faq/usage_faq.html#what-is-a-backend

Parameters

- **y_keys** (*iterable of strs*) – Keys of values regarded as y. If this is `None`, nothing is output to the graph.
- **x_key** (*str*) – Keys of values regarded as x. The default value is 'iteration'.
- **trigger** – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **postprocess** – Callback to postprocess the result dictionaries. Figure object, Axes object, and all plot data are passed to this callback in this order. This callback can modify the figure.
- **filename** (*str*) – Name of the figure file under the output directory. It can be a format string. For historical reasons `file_name` is also accepted as an alias of this argument.
- **marker** (*str*) – The marker used to plot the graph. Default is `'x'`. If `None` is given, it draws with no markers.
- **grid** (*bool*) – If `True`, set the axis grid on. The default value is `True`.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

static available()

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)

Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

Attributes

default_name

Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.VariableStatisticsPlot

class chainer.training.extensions.**VariableStatisticsPlot**(*targets*, *max_sample_size=1000*, *report_data=True*, *report_grad=True*, *plot_mean=True*, *plot_std=True*, *percentile_sigmas=(0, 0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87, 100)*, *trigger=(1, 'epoch')*, *filename='statistics.png'*, *figsize=None*, *marker=None*, *grid=True*)

Trainer extension to plot statistics for `Variable`s.

This extension collects statistics for a single `Variable`, a list of `Variable`s or similarly a single or a list of `Link`s containing one or more `Variable`s. In case multiple `Variable`s are found, the means are computed. The collected statistics are plotted and saved as an image in the directory specified by the `Trainer`.

Statistics include mean, standard deviation and percentiles.

This extension uses reservoir sampling to preserve memory, using a fixed size running sample. This means that collected items in the sample are discarded uniformly at random when the number of items becomes larger than the maximum sample size, but each item is expected to occur in the sample with equal probability.

Parameters

- **targets** (`Variable`, `Link` or list of either) – Parameters for which statistics are collected.
- **max_sample_size** (*int*) – Maximum number of running samples.
- **report_data** (*bool*) – If `True`, data (e.g. weights) statistics are plotted. If `False`, they are neither computed nor plotted.
- **report_grad** (*bool*) – If `True`, gradient statistics are plotted. If `False`, they are neither computed nor plotted.
- **plot_mean** (*bool*) – If `True`, means are plotted. If `False`, they are neither computed nor plotted.
- **plot_std** (*bool*) – If `True`, standard deviations are plotted. If `False`, they are neither computed nor plotted.
- **percentile_sigmas** (*float or tuple of floats*) – Percentiles to plot in the range $[0, 100]$.
- **trigger** – Trigger that decides when to save the plots as an image. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **filename** (*str*) – Name of the output image file under the output directory. For historical reasons `file_name` is also accepted as an alias of this argument.
- **figsize** (*tuple of int*) – Matlotlib `figsize` argument that specifies the size of the output image.
- **marker** (*str*) – Matplotlib `marker` argument that specified the marker style of the plots.

- **grid** (*bool*) – Matplotlib `grid` argument that specifies whether grids are rendered in in the plots or not.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

static available()

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

save_plot_using_module(*file_path*, *plt*)

serialize(*serializer*)

Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 100

trigger = (1, 'iteration')

chainer.training.extensions.DumpGraph

class chainer.training.extensions.**DumpGraph**(*root_name*, *filename='cg.dot'*, *variable_style=None*, *function_style=None*)
Trainer extension to dump a computational graph.

This extension dumps a computational graph. The graph is output in DOT language. If graphviz is available, this also renders and saves the image of the computational graph.

It only dumps a graph at the first invocation.

Note: The computational graph is not kept by default. This extension changes this behavior until the first invocation. **It is strongly recommended that you use it with the default trigger setting.**

The detailed behavior of this extension is as follows.

1. In its initializer, it turns on the `chainer.config.keep_graph_on_report` flag.
2. At the first iteration, it dumps the graph using the graph held by the reported variable.
3. After dumping the graph, it turns off the flag (if it was originally turned off) so that any variable reported afterward does not hold a computational graph.

When the `keep_graph_on_report` flag is turned on, the computational graph created by the updater is kept during the invocation of extensions. It will cause an unnecessarily large memory consumption when an extension also uses a large amount of memory, e.g. *Evaluator*.

With the default setting, the `DumpGraph` extension is called at the first iteration. Since *Evaluator* is not called at the first iteration in most cases, it does not cause any memory problem.

Parameters

- **root_name** (*str*) – Name of the root of the computational graph. The root variable is retrieved by this name from the observation dictionary of the trainer.
- **filename** (*str*) – Output file name. For historical reasons `out_name` is also accepted as an alias of this argument.
- **variable_style** (*dict*) – Dot node style for variables. Each variable is rendered by an octagon by default.
- **function_style** (*dict*) – Dot node style for functions. Each function is rendered by a rectangular by default.

See also:

See *build_computational_graph()* for the `variable_style` and `function_style` arguments.

Methods

__call__(*trainer*)
Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters trainer (`Trainer`) – Trainer object that calls this operator.

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(*trainer*)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters trainer (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)
Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (`Trainer`) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize (*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

trigger (*trainer*)
tuple() -> empty tuple tuple(iterable) -> tuple initialized from iterable's items

If the argument is a tuple, the return value is the same object.

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

Attributes

default_name = 'dump_graph'

name = None

priority = 100

Snapshot

These extensions provide features to take snapshots of models.

`chainer.training.extensions.snapshot`	Returns a trainer extension to take snapshots of the trainer.
`chainer.training.extensions.snapshot_object`	Returns a trainer extension to take snapshots of a given object.

chainer.training.extensions.snapshot

chainer.training.extensions.**snapshot** (*savefun=None, filename='snapshot_iter_{.updater.iteration}', *, target=None, condition=None, writer=None, snapshot_on_error=False, num_retain=-1, autoload=False*)
Returns a trainer extension to take snapshots of the trainer.

This extension serializes the trainer object and saves it to the output directory. It is used to support resuming the training loop from the saved state.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the *extend()* method of the trainer.

The default priority is -100, which is lower than that of most built-in extensions.

Note: This extension first writes the serialized object to a temporary file and then rename it to the target file name. Thus, if the program stops right before the renaming, the temporary file might be left in the output directory.

Parameters

- **savefun** – Function to save the trainer. It takes two arguments: the output file path and the trainer object. It is *chainer.serializers.save_npz()* by default. If `writer` is specified, this argument must be `None`.
- **filename** (*str*) – Name of the file into which the trainer is serialized. It can be a format string, where the trainer object is passed to the `str.format()` method.
- **target** – Object to serialize. If it is not specified, it will be the trainer object.
- **condition** – Condition object. It must be a callable object that returns boolean without any arguments. If it returns `True`, the snapshot will be done. If not, it will be skipped. The default is a function that always returns `True`.
- **writer** – Writer object. It must be a callable object. See below for the list of built-in writers. If `savefun` is other than `None`, this argument must be `None`. In that case, a *SimpleWriter* object instantiated with specified `savefun` argument will be used.
- **snapshot_on_error** (*bool*) – Whether to take a snapshot in case trainer loop has been failed.
- **num_retain** (*int*) – Number of snapshot files to retain through the cleanup. Must be a positive integer for any cleanup to take place. Automatic deletion of old snapshots only works when the filename is string.
- **autoload** (*bool*) – With this enabled, the extension automatically finds the latest snapshot and loads the data to the target. Automatic loading only works when the filename is a string. It is assumed that snapshots are generated by *chainer.serializers.save_npz()* .

Returns Snapshot extension object.

Using asynchronous writers

By specifying `writer` argument, writing operations can be made asynchronous, hiding I/O overhead of snapshots.

```
>>> from chainer.training import extensions
>>> writer = extensions.snapshot_writers.ProcessWriter()
>>> trainer.extend(extensions.snapshot(writer=writer), trigger=(1, 'epoch'))
```

To change the format, such as npz or hdf5, you can pass a saving function as `savefun` argument of the writer.

```
>>> from chainer.training import extensions
>>> from chainer import serializers
>>> writer = extensions.snapshot_writers.ProcessWriter(
...     savefun=serializers.save_npz)
>>> trainer.extend(extensions.snapshot(writer=writer), trigger=(1, 'epoch'))
```

This is the list of built-in snapshot writers.

- *chainer.training.extensions.snapshot_writers.SimpleWriter*
- *chainer.training.extensions.snapshot_writers.ThreadWriter*
- *chainer.training.extensions.snapshot_writers.ProcessWriter*
- *chainer.training.extensions.snapshot_writers.ThreadQueueWriter*
- *chainer.training.extensions.snapshot_writers.ProcessQueueWriter*

See also:

- *chainer.training.extensions.snapshot_object()*

chainer.training.extensions.snapshot_object

chainer.training.extensions.**snapshot_object**(*target*, *filename*, *savefun=None*, ***, *condition=None*, *writer=None*, *snapshot_on_error=False*, *num_retain=-1*, *autoload=False*)

Returns a trainer extension to take snapshots of a given object.

This extension serializes the given object and saves it to the output directory.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the *extend()* method of the trainer.

The default priority is -100, which is lower than that of most built-in extensions.

Parameters

- **target** – Object to serialize.
- **filename** (*str*) – Name of the file into which the object is serialized. It can be a format string, where the trainer object is passed to the `str.format()` method. For example, `'snapshot_{.updater.iteration}'` is converted to `'snapshot_10000'` at the 10,000th iteration.
- **savefun** – Function to save the object. It takes two arguments: the output file path and the object to serialize.
- **condition** – Condition object. It must be a callable object that returns boolean without any arguments. If it returns `True`, the snapshot will be done. If not, it will be skipped. The default is a function that always returns `True`.
- **writer** – Writer object. It must be a callable object. See below for the list of built-in writers. If `savefun` is other than `None`, this argument must be `None`. In that case, a *SimpleWriter* object instantiated with specified `savefun` argument will be used.
- **snapshot_on_error** (*bool*) – Whether to take a snapshot in case trainer loop has been failed.
- **num_retain** (*int*) – Number of snapshot files to retain through the cleanup. Must be a positive integer for any cleanup to take place. Automatic deletion of old snapshots only works when the filename is string.
- **autoload** (*bool*) – With this enabled, the extension automatically finds the latest snapshot and loads the data to the target. Automatic loading only works when the filename is a string.

Returns Snapshot extension object.

See also:

- *chainer.training.extensions.snapshot()*

Memory Release

These extensions provide features to release memories.

chainer.training.extensions.unchain_variables	Trainer extension to unchain all comptational graphs.

chainer.training.extensions.unchain_variables

class chainer.training.extensions.**unchain_variables**

Trainer extension to unchain all comptational graphs.

This extenstion unchains all comptational graphs after all extensions are run to release memory and to avoid memory leak. This extension can be used as a last resort when there is an extension that use a variable graph and cannot release the graph in itself. It observes the previous `chainer.config.keep_graph_on_report` flag. The extension is triggered when the flag is turned on.

Methods

__call__(*trainer*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters **trainer** (`Trainer`) – Trainer object that calls this operator.

finalize()

Finalizes the extension.

This method is called at the end of the training loop.

initialize(_)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of *Trainer* can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, *ExponentialShift* extension changes the optimizer's hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The *ExponentialShift* extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters **trainer** (`Trainer`) – Trainer object that runs the training loop.

on_error(*trainer*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **trainer** (Trainer) – Trainer object that runs the training loop.
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

serialize(*serializer*)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

trigger(_)
tuple() -> empty tuple tuple(iterable) -> tuple initialized from iterable's items

If the argument is a tuple, the return value is the same object.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

name = None

priority = 0

4.8.4 Triggers

A trigger is a callable object to decide when to process some specific event within the training loop. It takes a Trainer object as the argument, and returns True if some event should be fired.

It is mainly used to determine when to call an extension. It is also used to determine when to quit the training loop.

chainer.training.get_trigger	Gets a trigger object.
chainer.training.triggers.BestValueTrigger	Trigger invoked when specific value becomes best.
chainer.training.triggers.EarlyStoppingTrigger	Trigger for Early Stopping

Continued on next page

Table 44 – continued from previous page

chainer.training.triggers.IntervalTrigger	Trigger based on a fixed interval.
chainer.training.triggers.ManualScheduleTrigger	Trigger invoked at specified point(s) of iterations or epochs.
chainer.training.triggers.MaxValueTrigger	Trigger invoked when specific value becomes maximum.
chainer.training.triggers.MinValueTrigger	Trigger invoked when specific value becomes minimum.
chainer.training.triggers.OnceTrigger	Trigger based on the starting point of the iteration.
chainer.training.triggers.TimeTrigger	Trigger based on a fixed time interval.

chainer.training.get_trigger

chainer.training.**get_trigger**(*trigger*)

Gets a trigger object.

Trigger object is a callable that accepts a *Trainer* object as an argument and returns a boolean value. When it returns True, various kinds of events can occur depending on the context in which the trigger is used. For example, if the trigger is passed to the *Trainer* as the *stop trigger*, the training loop breaks when the trigger returns True. If the trigger is passed to the *extend()* method of a trainer, then the registered extension is invoked only when the trigger returns True.

This function returns a trigger object based on the argument. If `trigger` is already a callable, it just returns the trigger. If `trigger` is `None`, it returns a trigger that never fires. Otherwise, it passes the value to *IntervalTrigger*.

Parameters **trigger** – Trigger object. It can be either an already built trigger object (i.e., a callable object that accepts a trainer object and returns a bool value), or a tuple. In latter case, the tuple is passed to *IntervalTrigger*.

Returns `trigger` if it is a callable, otherwise a *IntervalTrigger* object made from `trigger`.

chainer.training.triggers.BestValueTrigger

class chainer.training.triggers.**BestValueTrigger**(*key*, *compare*, *trigger=(1, 'epoch')*)

Trigger invoked when specific value becomes best.

Parameters

- **key** (*str*) – Key of value.
- **compare** (*callable*) – Compare function which takes current best value and new value and returns whether new value is better than current best.
- **trigger** – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to *IntervalTrigger*.

Methods

__call__(*trainer*)

Decides whether the extension should be called on this iteration.

Parameters trainer (Trainer) – Trainer object that this trigger is associated with. The observation of this trainer is used to determine if the trigger should fire.

Returns True if the corresponding extension should be invoked in this iteration.

Return type bool

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.training.triggers.EarlyStoppingTrigger

class chainer.training.triggers.**EarlyStoppingTrigger**(*self, check_trigger=(1, 'epoch'), monitor='main/loss', patience=3, mode='auto', verbose=False, max_trigger=(100, 'epoch')*)

Trigger for Early Stopping

It can be used as a stop trigger of *Trainer* to realize *early stopping* technique.

This trigger works as follows. Within each *check interval* defined by the check_trigger argument, it monitors and accumulates the reported value at each iteration. At the end of each interval, it computes the mean of the accumulated values and compares it to the previous ones to maintain the *best* value. When it finds that the best value is not updated for some periods (defined by patience), this trigger fires.

Parameters

- **monitor** (*str*) – The metric you want to monitor
- **check_trigger** – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of <int>, 'epoch' or <int>, 'iteration' which is passed to *IntervalTrigger*.
- **patience** (*int*) – Counts to let the trigger be patient. The trigger will not fire until the condition is met for successive patience checks.
- **mode** (*str*) – 'max', 'min', or 'auto'. It is used to determine how to compare the monitored values.
- **verbose** (*bool*) – Enable verbose output. If verbose is true, you can get more information
- **max_trigger** – Upper bound of the number of training loops

Note: `patients` is also available as an alias of `patience` for historical reason.

Methods

`__call__`(*trainer*)
Decides whether the training loop should be stopped.

Parameters **trainer** (`Trainer`) – Trainer object that this trigger is associated with. The `observation` of this trainer is used to determine if the trigger should fire.

Returns `True` if the training loop should be stopped.

Return type bool

`get_training_length`()

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

chainer.training.triggers.IntervalTrigger

class `chainer.training.triggers.`**`IntervalTrigger`**(*period*, *unit*)
Trigger based on a fixed interval.

This trigger accepts iterations divided by a given interval. There are two ways to specify the interval: per iterations and epochs. *Iteration* means the number of updates, while *epoch* means the number of sweeps over the training dataset. Fractional values are allowed if the interval is a number of epochs; the trigger uses the *iteration* and *epoch_detail* attributes defined by the updater.

For the description of triggers, see *`get_trigger()`*.

Parameters

- **period** (*`int or float`*) – Length of the interval. Must be an integer if unit is `'iteration'`.
- **unit** (*`str`*) – Unit of the length specified by `period`. It must be either `'iteration'` or `'epoch'`.

Methods

`__call__`(*trainer*)
Decides whether the extension should be called on this iteration.

Parameters **`trainer`** (`Trainer`) – Trainer object that this trigger is associated with. The updater associated with this trainer is used to determine if the trigger should fire.

Returns True if the corresponding extension should be invoked in this iteration.

Return type bool

`get_training_length`()

`serialize`(*serializer*)

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

chainer.training.triggers.ManualScheduleTrigger

class `chainer.training.triggers.`**`ManualScheduleTrigger`**(*points*, *unit*)
Trigger invoked at specified point(s) of iterations or epochs.

This trigger accepts iterations or epochs indicated by given point(s). There are two ways to specify the point(s): iteration and epoch. `iteration` means the number of updates, while `epoch` means the number of sweeps over the training dataset. Fractional values are allowed if the point is a number of epochs; the trigger uses the `iteration` and `epoch_detail` attributes defined by the updater.

Parameters

- **`points`** (*`int, float, or list of int or float`*) – time of the trigger. Must be an integer or list of integer if unit is `'iteration'`.
- **`unit`** (*`str`*) – Unit of the time specified by `points`. It must be either `'iteration'` or `'epoch'`.

Variables

- **`finished`** (*`bool`*) – Flag that indicates whether or not this trigger will
- **`in the future. This flag is used to determine if the extension`** (*`fire`*) –
- **`be initialized after resume.`** (*`should`*) –

Methods

__call__(*trainer*)
Decides whether the extension should be called on this iteration.

Parameters **trainer** (Trainer) – Trainer object that this trigger is associated with. The updater associated with this trainer is used to determine if the trigger should fire.

Returns True if the corresponding extension should be invoked in this iteration.

Return type bool

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.training.triggers.MaxValueTrigger

class chainer.training.triggers.**MaxValueTrigger**(*key*, *trigger=(1, 'epoch')*)
Trigger invoked when specific value becomes maximum.

For example you can use this trigger to take snapshot on the epoch the validation accuracy is maximum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes maximum.
- **trigger** – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to *IntervalTrigger*.

Methods

__call__(*trainer*)
Decides whether the extension should be called on this iteration.

Parameters **trainer** (Trainer) – Trainer object that this trigger is associated with. The `observation` of this trainer is used to determine if the trigger should fire.

Returns `True` if the corresponding extension should be invoked in this iteration.

Return type bool

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.training.triggers.MinValueTrigger

class chainer.training.triggers.**MinValueTrigger**(*key*, *trigger=(1*, *'epoch')*)
Trigger invoked when specific value becomes minimum.

For example you can use this trigger to take snapshot on the epoch the validation loss is minimum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes minimum.
- **trigger** – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to *IntervalTrigger*.

Methods

__call__(*trainer*)
Decides whether the extension should be called on this iteration.

Parameters **trainer** (`Trainer`) – Trainer object that this trigger is associated with. The `observation` of this trainer is used to determine if the trigger should fire.

Returns `True` if the corresponding extension should be invoked in this iteration.

Return type bool

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.training.triggers.OnceTrigger

class chainer.training.triggers.**OnceTrigger**(*call_on_resume=False*)
Trigger based on the starting point of the iteration.

This trigger accepts only once at starting point of the iteration. There are two ways to specify the starting point: only starting point in whole iteration or called again when training resumed.

Parameters **call_on_resume** (*bool*) – Whether the extension is called again or not when restored from a snapshot. It is set to False by default.

Variables

- ***finished*** (*bool*) – Flag that indicates whether or not this trigger will
- **in the future. This flag is used to determine if the extension** (*fire*) –
- **be initialized after resume.** (*should*) –

Methods

__call__(*trainer*)
Call self as a function.

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

finished

chainer.training.triggers.TimeTrigger

class chainer.training.triggers.**TimeTrigger**(*period*)
Trigger based on a fixed time interval.

This trigger accepts iterations with a given interval time.

Parameters **period** (*float*) – Interval time. It is given in seconds.

Methods

__call__(*trainer*)
Call self as a function.

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.9 Datasets

4.9.1 Dataset Abstraction (chainer.dataset)

Chainer supports a common interface for training and validation of datasets. The dataset support consists of three components: datasets, iterators, and batch conversion functions.

Dataset represents a set of examples. The interface is only determined by combination with iterators you want to use on it. The built-in iterators of Chainer require the dataset to support `__getitem__` and `__len__` methods. In particular, the `__getitem__` method should support indexing by both an integer and a slice. We can easily support slice indexing by inheriting `DatasetMixin`, in which case users only have to implement `get_example()` method for indexing. Basically, datasets are considered as *stateless* objects, so that we do not need to save the dataset as a checkpoint of the training procedure.

Iterator iterates over the dataset, and at each iteration, it yields a mini-batch of examples as a list. Iterators should support the `Iterator` interface, which includes the standard iterator protocol of Python. Iterators manage where to read next, which means they are *stateful*.

Batch conversion function converts the mini-batch into arrays to feed to the neural nets. They are also responsible to send each array to an appropriate device. Chainer currently provides two implementations:

- `concat_examples()` is a plain implementation which is used as the default choice.
- `ConcatWithAsyncTransfer` is a variant which is basically same as `concat_examples()` except that it overlaps other GPU computations and data transfer for the next iteration.

These components are all customizable, and designed to have a minimum interface to restrict the types of datasets and ways to handle them. In most cases, though, implementations provided by Chainer itself are enough to cover the usages.

Chainer also has a light system to download, manage, and cache concrete examples of datasets. All datasets managed through the system are saved under *the dataset root directory*, which is determined by the `CHAINER_DATASET_ROOT` environment variable, and can also be set by the `set_dataset_root()` function.

Dataset Representation

See *Dataset Examples (chainer.datasets)* for dataset implementations.

`chainer.dataset.DatasetMixin`	Default implementation of dataset indexing.

chainer.dataset.DatasetMixin

class chainer.dataset.**DatasetMixin**

Default implementation of dataset indexing.

DatasetMixin provides the `__getitem__()` operator. The default implementation uses `get_example()` to extract each example, and combines the results into a list. This mixin makes it easy to implement a new dataset that does not support efficient slicing.

Dataset implementation using DatasetMixin still has to provide the `__len__()` operator explicitly.

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (`int, slice, list or numpy.ndarray`) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]    # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
```

(continues on next page)

(continued from previous page)

```
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

get_example(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Tabular Dataset Representation

chainer.dataset.TabularDataset	An abstract class that represents tabular dataset.

chainer.dataset.TabularDataset

class chainer.dataset.**TabularDataset**
An abstract class that represents tabular dataset.

This class represents a tabular dataset. In a tabular dataset, all examples have the same number of elements. For example, all examples of the dataset below have three elements (`a[i]`, `b[i]`, and `c[i]`).

	a	b	c
0	a[0]	b[0]	c[0]
1	a[1]	b[1]	c[1]
2	a[2]	b[2]	c[2]
3	a[3]	b[3]	c[3]

Since an example can be represented by both tuple and dict (`(a[i], b[i], c[i])` and `{'a': a[i], 'b': b[i], 'c': c[i]}`), this class uses *mode* to indicate which representation will be used. If there is only one column, an example also can be represented by a value (`a[i]`). In this case, *mode* is `None`.

An inheritance should implement `__len__()`, *keys*, *mode* and `get_examples()`.

```
>>> import numpy as np
>>>
>>> from chainer import dataset
>>>
>>> class MyDataset(dataset.TabularDataset):
...
...     def __len__(self):
...         return 4
...
...     @property
...     def keys(self):
...          return ('a', 'b', 'c')
...
...     @property
...     def mode(self):
...          return tuple
...
...     def get_examples(self, indices, key_indices):
...          data = np.arange(12).reshape((4, 3))
...          if indices is not None:
...              data = data[indices]
...          if key_indices is not None:
...              data = data[:, list(key_indices)]
...          return tuple(data.transpose())
...
>>> dataset = MyDataset()
>>> len(dataset)
4
>>> dataset.keys
('a', 'b', 'c')
>>> dataset.as_tuple()[0]
(0, 1, 2)
>>> sorted(dataset.as_dict()[0].items())
[('a', 0), ('b', 1), ('c', 2)]
>>>
>>> view = dataset.slice[[3, 2], ('c', 0)]
>>> len(view)
2
>>> view.keys
('c', 'a')
>>> view.as_tuple()[1]
(8, 6)
>>> sorted(view.as_dict()[1].items())
[('a', 6), ('c', 8)]
```

Methods

`__getitem__`(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__`()
Returns the number of data points.

`__iter__`()

`as_dict`()
Return a view with dict mode.

Returns A view whose *mode* is `dict`.

`as_tuple`()
Return a view with tuple mode.

Returns A view whose *mode* is `tuple`.

`concat`(**datasets*)
Stack datasets along rows.

Parameters **datasets** (iterable of *TabularDataset*) – Datasets to be concatenated. All datasets must have the same *keys*.

Returns A concatenated dataset.

`convert`(*data*)
Convert fetched data.

This method takes data fetched by *fetch()* and pre-process them before passing them to models. The default behaviour is converting each column into an ndarray. This behaviour can be overridden by *with_converter()*. If the dataset is constructed by *concat()* or *join()*, the converter of the first dataset is used.

Parameters **data** (*tuple or dict*) – Data from *fetch()*.

Returns A tuple or dict. Each value is an ndarray.

fetch()
Fetch data.

This method fetches all data of the dataset/view. Note that this method returns a column-major data (i.e. `([a[0], ..., a[3]], ..., [c[0], ... c[3]])`, `{'a': [a[0], ..., a[3]], ..., 'c': [c[0], ..., c[3]]}`, or `[a[0], ..., a[3]]`).

Returns If *mode* is `tuple`, this method returns a tuple of lists/arrays. If *mode* is `dict`, this method returns a dict of lists/arrays.

get_example(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

get_examples(*indices*, *key_indices*)
Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is `None`, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is `None`, it indicates all columns.

Returns tuple of lists/arrays

join(**datasets*)
Stack datasets along columns.

Parameters **datasets** (iterable of *TabularDataset*) – Datasets to be concatenated. All datasets must have the same length

Returns A joined dataset.

transform(*keys*, *transform*)
Apply a transform to each example.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform** (*callable*) – A callable that takes an example and returns transformed example. *mode* of transformed dataset is determined by the transformed examples.

Returns A transfromed dataset.

transform_batch(*keys*, *transform_batch*)
Apply a transform to examples.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform_batch** (*callable*) – A callable that takes examples and returns transformed examples. *mode* of transformed dataset is determined by the transformed examples.

Returns A transfromed dataset.

with_converter(*converter*)
Override the behaviour of `convert()`.

This method overrides `convert()`.

Parameters **converter** (*callable*) – A new converter.

Returns A dataset with the new converter.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

keys
Names of columns.

A tuple of strings that indicate the names of columns.

mode
Mode of representation.

This indicates the type of value returned by `fetch()` and `__getitem__()`. `tuple`, `dict`, and `None` are supported.

slice
Get a slice of dataset.

Parameters

- **indices** (*list/array of ints/bools or slice*) – Requested rows.
- **keys** (*tuple of ints/strs or int or str*) – Requested columns.

Returns A view of specified range.

Tabular Dataset Helpers

`chainer.dataset.tabular.DelegateDataset`	A helper class to implement a TabularDataset.
`chainer.dataset.tabular.from_data`	Create a TabularDataset from lists/arrays/callables.

chainer.dataset.tabular.DelegateDataset

class chainer.dataset.tabular.**DelegateDataset**(*dataset*)

A helper class to implement a TabularDataset.

This class wraps an instance of *TabularDataset* and provides methods of *TabularDataset*. This class is useful to create a custom dataset class by inheriting it.

```
>>> import numpy as np
>>>
>>> from chainer.dataset import tabular
>>>
>>> class MyDataset(tabular.DelegateDataset):
...
...     def __init__(self):
...         super().__init__(tabular.from_data((
...             ('a', np.arange(10)),
...             ('b', self.get_b),
...             ('c', [3, 1, 4, 5, 9, 2, 6, 8, 7, 0]),
...             (('d', 'e'), self.get_de))))
...
...     def get_b(self, i):
...         return 'b[{}]'.format(i)
...
...     def get_de(self, i):
...         return {'d': 'd[{}]'.format(i), 'e': 'e[{}]'.format(i)}
...
>>> dataset = MyDataset()
>>> len(dataset)
10
>>> dataset.keys
('a', 'b', 'c', 'd', 'e')
>>> dataset[0]
(0, 'b[0]', 3, 'd[0]', 'e[0]')
```

Parameters **dataset** (chainer.dataset.TabularDataset) – An underlying dataset.

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
```

(continues on next page)

(continued from previous page)

```
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__`()
Returns the number of data points.

`__iter__`()

`as_dict`()
Return a view with dict mode.

Returns A view whose *`mode`* is `dict`.

`as_tuple`()
Return a view with tuple mode.

Returns A view whose *`mode`* is `tuple`.

`concat`(**datasets*)
Stack datasets along rows.

Parameters **datasets** (iterable of `TabularDataset`) – Datasets to be concatenated. All datasets must have the same *`keys`*.

Returns A concatenated dataset.

`convert`(*data*)
Convert fetched data.

This method takes data fetched by *`fetch()`* and pre-process them before passing them to models. The default behaviour is converting each column into an ndarray. This behaviour can be overridden by *`with_converter()`*. If the dataset is constructed by *`concat()`* or *`join()`*, the converter of the first dataset is used.

Parameters **data** (*tuple or dict*) – Data from *`fetch()`*.

Returns A tuple or dict. Each value is an ndarray.

`fetch`()
Fetch data.

This method fetches all data of the dataset/view. Note that this method returns a column-major data (i.e. `([a[0], ..., a[3]], ..., [c[0], ... c[3]])`, `{'a': [a[0], ..., a[3]], ..., 'c': [c[0], ..., c[3]]}`, or `[a[0], ..., a[3]]`).

Returns If *mode* is tuple, this method returns a tuple of lists/arrays. If *mode* is dict, this method returns a dict of lists/arrays.

get_example (*i*)
Returns the i-th example.

Implementations should override it. It should raise IndexError if the index is invalid.

Parameters i (*int*) – The index of the example.

Returns The i-th example.

get_examples (*indices*, *key_indices*)
Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is None, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is None, it indicates all columns.

Returns tuple of lists/arrays

join (**datasets*)
Stack datasets along columns.

Parameters datasets (iterable of TabularDataset) – Datasets to be concatenated. All datasets must have the same length

Returns A joined dataset.

transform (*keys*, *transform*)
Apply a transform to each example.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform** (*callable*) – A callable that takes an example and returns transformed example. *mode* of transformed dataset is determined by the transformed examples.

Returns A transfromed dataset.

transform_batch (*keys*, *transform_batch*)
Apply a transform to examples.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform_batch** (*callable*) – A callable that takes examples and returns transformed examples. *mode* of transformed dataset is determined by the transformed examples.

Returns A transfromed dataset.

with_converter (*converter*)
Override the behaviour of *convert()*.

This method overrides *convert()*.

Parameters converter (*callable*) – A new converter.

Returns A dataset with the new converter.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

keys
Names of columns.

A tuple of strings that indicate the names of columns.

mode
Mode of representation.

This indicates the type of value returned by *fetch()* and *__getitem__()*. tuple, dict, and None are supported.

slice
Get a slice of dataset.

Parameters

- **indices** (*list/array of ints/bools or slice*) – Requested rows.
- **keys** (*tuple of ints/strs or int or str*) – Requested columns.

Returns A view of specified range.

chainer.dataset.tabular.from_data

chainer.dataset.tabular.**from_data**(*data*, *size=None*)
Create a TabularDataset from lists/arrays/callables.

```
>>> from chainer.dataset import tabular
>>>
>>> dataset = tabular.from_data([0, 1, 2])
>>> dataset[0]
0
>>> dataset = tabular.from_data(([0, 1, 2], [3, 4, 5]))
>>> dataset[0]
(0, 3)
>>> dataset = tabular.from_data((('a', [0, 1, 2]), ('b', [3, 4, 5])))
>>> dataset.keys
('a', 'b')
>>> dataset[0]
```

(continues on next page)

(continued from previous page)

```
(0, 3)
>>> dataset = tabular.from_data({'a': [0, 1, 2], 'b': [3, 4, 5]})
>>> sorted(dataset[0].items())
[('a', 0), ('b', 3)]
>>> dataset = tabular.from_data(('a', lambda i: i * i), size=10)
>>> dataset[5]
25
```

Parameters

- **data** (*list, array, tuple, or dict*) – Data in following format.
  - *list/array*
  - *(str, list/array/callable)*
  - *((str, . . .), callable)*
  - *((list/array)/(str, list/array/callable) /((key, . . .), callable), . . .)*
  - *{str: (list/array/callable)/(str, . . .): callable, . . . }*
- **size** (*int*) – The length of the dataset. This argument is required when no lists/arrays exist in `data`.

Returns A `TabularDataset`.

Iterator Interface

See *Iterator* for dataset iterator implementations.

`chainer.dataset.Iterator`	Base class of all dataset iterators.

chainer.dataset.Iterator

class chainer.dataset.**Iterator**

Base class of all dataset iterators.

Iterator iterates over the dataset, yielding a minibatch at each iteration. Minibatch is a list of examples. Each implementation should implement an iterator protocol (e.g., the `__next__()` method).

Note that, even if the iterator supports setting the batch size, it does not guarantee that each batch always contains the same number of examples. For example, if you let the iterator to stop at the end of the sweep, the last batch may contain a fewer number of examples.

The interface between the iterator and the underlying dataset is not fixed, and up to the implementation.

Each implementation should provide the following attributes (not needed to be writable).

- `batch_size`: Number of examples within each minibatch.
- `epoch`: Number of completed sweeps over the dataset.
- `epoch_detail`: Floating point number version of the epoch. For example, if the iterator is at the middle of the dataset at the third epoch, then this value is 2.5.
- `previous_epoch_detail`: The value of `epoch_detail` at the previous iteration. This value is `None` before the first iteration.

- `is_new_epoch`: `True` if the epoch count was incremented at the last update.

Each implementation should also support serialization to resume/suspend the iteration.

Methods

`__enter__`()

`__exit__`(*exc_type*, *exc_value*, *traceback*)

`__next__`()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

`__iter__`()
Returns self.

`finalize`()
Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

This method can be called multiple times.

`next`()
Python2 alternative of `__next__`.

It calls `__next__()` by default.

`serialize`(*serializer*)
Serializes the internal state of the iterator.

This is a method to support the serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serialize what is set manually by users such as the batch size.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

Batch Conversion Function

chainer.dataset.Converter	Base class of converters.
chainer.dataset.converter	Decorator to make a converter.
chainer.dataset.concat_examples	Converter to wrap a callable with arbitrary arguments.
chainer.dataset.ConcatWithAsyncTransfer	Interface to concatenate data and transfer them to GPU asynchronously.
chainer.dataset.to_device	Send an array to a given device.

chainer.dataset.Converter

class chainer.dataset.**Converter**

Base class of converters.

Converters receive batched data retrieved from iterators and perform arbitrary transforms as well as device transfer.

Implementation should override the `__call__` method.

See also:

chainer.dataset.converter() — a decorator to turn a converter function into a `Converter` instance.

Methods

__call__(*batch*, *device*)

Performs conversion.

Parameters

- **batch** – A batch. The type and value are arbitrary, depending on usage.
- **device** (`Device`) – Device to which the converter is expected to send the batch.

Returns: A converted batch.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

chainer.dataset.converter

chainer.dataset.**converter**()

Decorator to make a converter.

This decorator turns a converter function into a *chainer.dataset.Converter* class instance, which also is a callable. This is required to use the converter function from an old module that does not support *chainer.backend.Device* instances (See the **Device argument conversion** section below).

Requirements of the target function

The target converter function must accept two positional arguments: a batch and a device, and return a converted batch.

The type of the device argument is *chainer.backend.Device*.

The types and values of the batches (the first argument and the return value) are not specified: they depend on how the converter is used (e.g. by updaters).

Example

```
>>> @chainer.dataset.converter()
... def custom_converter(batch, device):
...     assert isinstance(device, chainer.backend.Device)
...     # do something with batch...
...     return device.send(batch)
```

Device argument conversion

For backward compatibility, the decorator wraps the function so that if the converter is called with the device argument with `int` type, it is converted to a *chainer.backend.Device* instance before calling the original function. The `int` value indicates the CUDA device of the cupy backend.

Without the decorator, the converter cannot support ChainerX devices. If the batch were requested to be converted to ChainerX with such converters, `RuntimeError` will be raised.

chainer.dataset.concat_examples

chainer.dataset.**concat_examples = <chainer.dataset.convert._ArbitraryCallableConverter obje**

Converter to wrap a callable with arbitrary arguments.

This class accepts arbitrary arguments and pass-through to the underlying callable, with device argument replaced.

chainer.dataset.ConcatWithAsyncTransfer

class chainer.dataset.**ConcatWithAsyncTransfer**(*stream=None*, *compute_stream=None*)

Interface to concatenate data and transfer them to GPU asynchronously.

It enables to transfer next batch of input data to GPU while GPU is running kernels for training using current batch of input data.

An instance of this class is mainly intended to be used as a converter function of an updater like below.

```
from chainer.dataset import convert
...
updater = chainer.training.updaters.StandardUpdater(
               ...,
               converter=convert.ConcatWithAsyncTransfer(),
               ...)
```

Parameters

- **stream** (*cupy.cuda.Stream*) – CUDA stream. If `None`, a stream is automatically created on the first call. Data transfer operation is launched asynchronously using the stream.
- **compute_stream** (*cupy.cuda.Stream*) – CUDA stream used for compute kernels. If not `None`, CUDA events are created/used to avoid global synchronization and overlap execution of compute kernels and data transfers as much as possible. If `None`, global synchronization is used instead.

Methods

__call__(*batch*, *device=None*, *padding=None*)
Concatenate data and transfer them to GPU asynchronously.

See also `chainer.dataset.concat_examples()`.

Parameters

- **batch** (*list*) – A list of examples.
- **device** (*int*) – Device ID to which each array is sent.
- **padding** – Scalar value for extra elements.

Returns Array, a tuple of arrays, or a dictionary of arrays. The type depends on the type of each example in the batch.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.dataset.to_device

chainer.dataset.**to_device**(*device*, *x*)
Send an array to a given device.

This method sends a given array to a given device. This method is used in *concat_examples()*. You can also use this method in a custom converter method used in *Updater* and *Extension* such as *StandardUpdater* and *Evaluator*.

See also *chainer.dataset.concat_examples()*.

Parameters

- **device** (*None or int or device specifier*) – A device to which an array is sent. If it is a negative integer, an array is sent to CPU. If it is a positive integer, an array is sent to GPU with the given ID. If it is‘‘None‘‘, an array is left in the original device. Also, any of device specifiers described at `DeviceId` is accepted.
- **x** (*N-dimensional array*) – An array to send.

Returns Converted array.

Dataset Management

chainer.dataset.get_dataset_root	Gets the path to the root directory to download and cache datasets.
chainer.dataset.set_dataset_root	Sets the root directory to download and cache datasets.
chainer.dataset.cached_download	Downloads a file and caches it.
chainer.dataset.cache_or_load_file	Caches a file if it does not exist, or loads it otherwise.

chainer.dataset.get_dataset_root

`chainer.dataset.`**`get_dataset_root`**`()`

Gets the path to the root directory to download and cache datasets.

Returns The path to the dataset root directory.

Return type str

chainer.dataset.set_dataset_root

`chainer.dataset.`**`set_dataset_root`**`(`*path*`)`

Sets the root directory to download and cache datasets.

There are two ways to set the dataset root directory. One is by setting the environment variable `CHAINER_DATASET_ROOT`. The other is by using this function. If both are specified, one specified via this function is used. The default dataset root is `$HOME/.chainer/dataset`.

Parameters **path** (*str*) – Path to the new dataset root directory.

chainer.dataset.cached_download

`chainer.dataset.`**`cached_download`**`(`*url*`)`

Downloads a file and caches it.

It downloads a file from the URL if there is no corresponding cache. After the download, this function stores a cache to the directory under the dataset root (see *set_dataset_root()*). If there is already a cache for the given URL, it just returns the path to the cache without downloading the same file.

Note: This function raises `OSError` when it fails to create the cache directory. In older version, it raised `RuntimeError`.

Parameters **url** (*str*) – URL to download from.

Returns Path to the downloaded file.

Return type str

chainer.dataset.cache_or_load_file

`chainer.dataset.cache_or_load_file(path, creator, loader)`

Caches a file if it does not exist, or loads it otherwise.

This is a utility function used in dataset loading routines. The `creator` creates the file to given path, and returns the content. If the file already exists, the `loader` is called instead, and it loads the file and returns the content.

Note that the path passed to the creator is temporary one, and not same as the path given to this function. This function safely renames the file created by the creator to a given path, even if this function is called simultaneously by multiple threads or processes.

Parameters

- **path** (*str*) – Path to save the cached file.
- **creator** – Function to create the file and returns the content. It takes a path to temporary place as the argument. Before calling the creator, there is no file at the temporary path.
- **loader** – Function to load the cached file and returns the content.

Returns It returns the returned values by the creator or the loader.

4.9.2 Dataset Examples (`chainer.datasets`)

The most basic *dataset* implementation is an array. Both NumPy and CuPy arrays can be used directly as datasets.

In many cases, though, the simple arrays are not enough to write the training procedure. In order to cover most of such cases, Chainer provides many built-in implementations of datasets.

These built-in datasets are divided into two groups. One is a group of general datasets. Most of them are wrapper of other datasets to introduce some structures (e.g., tuple or dict) to each data point. The other one is a group of concrete, popular datasets. These concrete examples use the downloading utilities in the `chainer.dataset` module to cache downloaded and converted datasets.

4.9.3 General Datasets

General datasets are further divided into four types.

The first one is `DictDataset` and `TupleDataset`, both of which combine other datasets and introduce some structures on them.

The second one is `ConcatenatedDataset` and `SubDataset`. `ConcatenatedDataset` represents a concatenation of existing datasets. It can be used to merge datasets and make a larger dataset. `SubDataset` represents a subset of an existing dataset. It can be used to separate a dataset for hold-out validation or cross validation. Convenient functions to make random splits are also provided.

The third one is *TransformDataset*, which wraps around a dataset by applying a function to data indexed from the underlying dataset. It can be used to modify behavior of a dataset that is already prepared.

The last one is a group of domain-specific datasets. Currently, implementations for datasets of images (*ImageDataset*, *LabeledImageDataset*, etc.) and text (*TextDataset*) are provided.

DictDataset

chainer.datasets.DictDataset	Dataset of a dictionary of datasets.

chainer.datasets.DictDataset

class chainer.datasets.**DictDataset**(***datasets*)

Dataset of a dictionary of datasets.

It combines multiple datasets into one dataset. Each example is represented by a dictionary mapping a key to an example of the corresponding dataset.

Parameters **datasets** – Underlying datasets. The keys are used as the keys of each example. All datasets must have the same length.

Methods

__getitem__(*index*)

__len__()

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

TupleDataset

chainer.datasets.TupleDataset	Dataset of tuples from multiple equal-length datasets.

chainer.datasets.TupleDataset

class chainer.datasets.**TupleDataset**(**datasets*)

Dataset of tuples from multiple equal-length datasets.

A `TupleDataset` combines multiple equal-length datasets into a single dataset of tuples. The `i`-th tuple contains the `i`-th example from each of the argument datasets, in the same order that they were supplied.

Recall that in Chainer, a dataset is defined as an iterable that supports both `__getitem__` and `__len__`. The `__getitem__` method should support indexing by both an integer and a slice.

As an example, consider creating a `TupleDataset` from two argument datasets `d1 = [8, 0, 5, 1]` and `d2 = [3, 1, 7, 4]` as `tuple_dataset = TupleDataset(d1, d2)`. The `tuple_dataset` will then contain the examples `(8, 3), (0, 1), (5, 7), (1, 4)`. Note that this behavior is similar to that of the built-in `zip()` function.

Parameters **`datasets`** – Underlying datasets that will be aggregated. Each dataset must be an iterable that implements `__getitem__` and `__len__`. The `j`-th dataset will be used for the `j`-th item of each example tuple. All datasets must have the same length.

Methods

`__getitem__`(*index*)

`__len__`()

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

ConcatenatedDataset

`chainer.datasets.ConcatenatedDataset`	Dataset which concatenates some base datasets.

chainer.datasets.ConcatenatedDataset

class `chainer.datasets.`**`ConcatenatedDataset`**(**datasets*)
Dataset which concatenates some base datasets.

This dataset wraps some base datasets and works as a concatenated dataset. For example, if a base dataset with 10 samples and another base dataset with 20 samples are given, this dataset works as a dataset which has 30 samples.

Parameters **`datasets`** – The underlying datasets. Each dataset has to support *`__len__()`* and *`__getitem__()`*.

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()

Returns the number of data points.

get_example(*i*)

Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

SubDataset

chainer.datasets.SubDataset	Subset of a base dataset.
chainer.datasets.split_dataset	Splits a dataset into two subsets.
chainer.datasets.split_dataset_random	Splits a dataset into two subsets randomly.
chainer.datasets.get_cross_validation_datasets	Creates a set of training/test splits for cross validation.
chainer.datasets.get_cross_validation_datasets_random	Creates a set of training/test splits for cross validation randomly.

chainer.datasets.SubDataset

class chainer.datasets.**SubDataset**(*dataset*, *start*, *finish*, *order=None*)
Subset of a base dataset.

SubDataset defines a subset of a given base dataset. The subset is defined as an interval of indexes, optionally with a given permutation.

If `order` is given, then the `i`-th example of this dataset is the `order[start + i]`-th example of the base dataset, where `i` is a non-negative integer. If `order` is not given, then the `i`-th example of this dataset is the `start + i`-th example of the base dataset. Negative indexing is also allowed: in this case, the term `start + i` is replaced by `finish + i`.

SubDataset is often used to split a dataset into training and validation subsets. The training set is used for training, while the validation set is used to track the generalization performance, i.e. how the learned model works well on unseen data. We can tune hyperparameters (e.g. number of hidden units, weight initializers, learning rate, etc.) by comparing the validation performance. Note that we often use another set called test set to measure the quality of the tuned hyperparameter, which can be made by nesting multiple SubDatasets.

There are two ways to make training-validation splits. One is a single split, where the dataset is split just into two subsets. It can be done by *split_dataset()* or *split_dataset_random()*. The other one is a k-fold cross validation, in which the dataset is divided into k subsets, and k different splits are generated using each of the k subsets as a validation set and the rest as a training set. It can be done by *get_cross_validation_datasets()*.

Parameters

- **dataset** – Base dataset.
- **start** (*int*) – The first index in the interval.
- **finish** (*int*) – The next-to-the-last index in the interval.
- **order** (*sequence of ints*) – Permutation of indexes in the base dataset. If this is `None`, then the ascending order of indexes is used.

Methods

__getitem__(*index*)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

get_example(*i*)
Returns the i-th example.

Implementations should override it. It should raise IndexError if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.datasets.split_dataset

chainer.datasets.**split_dataset**(*dataset*, *split_at*, *order=None*)
Splits a dataset into two subsets.

This function creates two instances of *SubDataset*. These instances do not share any examples, and they together cover all examples of the original dataset.

Parameters

- **dataset** – Dataset to split.
- **split_at** (*int*) – Position at which the base dataset is split.
- **order** (*sequence of ints*) – Permutation of indexes in the base dataset. See the documentation of *SubDataset* for details.

Returns Two *SubDataset* objects. The first subset represents the examples of indexes `order[:split_at]` while the second subset represents the examples of indexes `order[split_at:]`.

Return type tuple

chainer.datasets.split_dataset_random

chainer.datasets.**split_dataset_random**(*dataset*, *first_size*, *seed=None*)
Splits a dataset into two subsets randomly.

This function creates two instances of *SubDataset*. These instances do not share any examples, and they together cover all examples of the original dataset. The split is automatically done randomly.

Parameters

- **dataset** – Dataset to split.
- **first_size** (*int*) – Size of the first subset.
- **seed** (*int*) – Seed the generator used for the permutation of indexes. If an integer being convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the given dataset always belongs to a specific subset. If `None`, the permutation is changed randomly.

Returns Two *SubDataset* objects. The first subset contains `first_size` examples randomly chosen from the dataset without replacement, and the second subset contains the rest of the dataset.

Return type tuple

chainer.datasets.get_cross_validation_datasets

`chainer.datasets.get_cross_validation_datasets(dataset, n_fold, order=None)`

Creates a set of training/test splits for cross validation.

This function generates `n_fold` splits of the given dataset. The first part of each split corresponds to the training dataset, while the second part to the test dataset. No pairs of test datasets share any examples, and all test datasets together cover the whole base dataset. Each test dataset contains almost same number of examples (the numbers may differ up to 1).

Parameters

- **dataset** – Dataset to split.
- **n_fold** (*int*) – Number of splits for cross validation.
- **order** (*sequence of ints*) – Order of indexes with which each split is determined. If it is `None`, then no permutation is used.

Returns List of dataset splits.

Return type list of tuples

chainer.datasets.get_cross_validation_datasets_random

`chainer.datasets.get_cross_validation_datasets_random(dataset, n_fold, seed=None)`

Creates a set of training/test splits for cross validation randomly.

This function acts almost same as `get_cross_validation_dataset()`, except automatically generating random permutation.

Parameters

- **dataset** – Dataset to split.
- **n_fold** (*int*) – Number of splits for cross validation.
- **seed** (*int*) – Seed the generator used for the permutation of indexes. If an integer beging convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the given dataset always belongs to a specific subset. If `None`, the permutation is changed randomly.

Returns List of dataset splits.

Return type list of tuples

TransformDataset

`chainer.datasets.TransformDataset`	Dataset that indexes the base dataset and transforms the data.

chainer.datasets.TransformDataset

`class chainer.datasets.TransformDataset(dataset, transform)`

Dataset that indexes the base dataset and transforms the data.

This dataset wraps the base dataset by modifying the behavior of the base dataset's `__getitem__()`. Arrays returned by `__getitem__()` of the base dataset with an integer as an argument are transformed by the given function `transform`. Also, *__len__()* returns the integer returned by the base dataset's *__len__()*.

The function `transform` takes, as an argument, `in_data`, which is the output of the base dataset's *__getitem__()*, and returns the transformed arrays as output. Please see the following example. Since `in_data` directly refers to the item in the dataset, take care that `transform` not modify it. For example, note that the line *img = img - 0.5* bellow is correct since it makes a copy of *img*. However, it would be incorrect to use *img -= 0.5* since that would update the contents of the item in the dataset in place, corrupting it.

```
>>> from chainer.datasets import get_mnist
>>> from chainer.datasets import TransformDataset
>>> dataset, _ = get_mnist()
>>> def transform(in_data):
...     img, label = in_data
...     img = img - 0.5  # scale to [-0.5, 0.5]
...     return img, label
>>> dataset = TransformDataset(dataset, transform)
```

Parameters

- **dataset** – The underlying dataset. The index of this dataset corresponds to the index of the base dataset. This object needs to support functions *__getitem__()* and *__len__()* as described above.
- **transform** (*callable*) – A function that is called to transform values returned by the underlying dataset's *__getitem__()*.

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
```

(continues on next page)

(continued from previous page)

```
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

get_example(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

ImageDataset

chainer.datasets.ImageDataset	Dataset of images built from a list of paths to image files.
chainer.datasets.ZippedImageDataset	Dataset of images built from a zip file.
chainer.datasets.MultiZippedImageDataset	Dataset of images built from a list of paths to zip files.

chainer.datasets.ImageDataset

class chainer.datasets.**ImageDataset**(*paths*, *root='.'*, *dtype=None*)
Dataset of images built from a list of paths to image files.

This dataset reads an external image file on every call of the *__getitem__()* operator. The paths to the image to retrieve is given as either a list of strings or a text file that contains paths in distinct lines.

Each image is automatically converted to arrays of shape `channels, height, width`, where `channels` represents the number of channels in each pixel (e.g., 1 for grey-scale images, and 3 for RGB-color images).

Note: This dataset requires the Pillow package being installed. In order to use this dataset, install Pillow (e.g. by using the command `pip install Pillow`). Be careful to prepare appropriate libraries for image formats you want to use (e.g. libpng for PNG images, and libjpeg for JPG images).

Warning: You are responsible for preprocessing the images before feeding them to a model. For example, if your dataset contains both RGB and grayscale images, make sure that you convert them to the same format. Otherwise you will get errors because the input dimensions are different for RGB and grayscale images.

Parameters

- **paths** (*str or list of strs*) – If it is a string, it is a path to a text file that contains paths to images in distinct lines. If it is a list of paths, the `i`-th element represents the path to the `i`-th image. In both cases, each path is a relative one from the root path given by another argument.
- **root** (*str*) – Root directory to retrieve images from.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__`()
Returns the number of data points.

`get_example`(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

chainer.datasets.ZippedImageDataset

class `chainer.datasets.`**`ZippedImageDataset`**(*zipfilename*, *dtype=None*)
Dataset of images built from a zip file.

This dataset reads an external image file in the given zipfile. The zipfile shall contain only image files. This shall be able to replace ImageDataset and works better on NFS and other networked file systems. If zipfile becomes too large you may consider `MultiZippedImageDataset` as a handy alternative.

Known issue: pickle and unpickle on same process may cause race condition on ZipFile. Pickle of this class is expected to be sent to different processess via ChainerMN.

Parameters

- **zipfilename** (*str*) – a string to point zipfile path
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]    # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()

Returns the number of data points.

get_example(*i_or_filename*)

Returns the i-th example.

Implementations should override it. It should raise IndexError if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.datasets.MultiZippedImageDataset

class chainer.datasets.**MultiZippedImageDataset**(*zipfilenames*, *dtype=None*)
Dataset of images built from a list of paths to zip files.

This dataset reads an external image file in given zipfiles. The zipfiles shall contain only image files. This shall be able to replace ImageDataset and works better on NFS and other networked file systems. The user shall find good balance between zipfile size and number of zipfiles (e.g. granularity)

Parameters

- **zipfilenames** (*list of strings*) – List of zipped archive filename.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).

Methods

__getitem__(*index*)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__`()
Returns the number of data points.

`get_example`(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

LabeledImageDataset

`chainer.datasets.LabeledImageDataset`	Dataset of image and label pairs built from a list of paths and labels.
`chainer.datasets.LabeledZippedImageDataset`	Dataset of zipped image and label pairs.

chainer.datasets.LabeledImageDataset

class chainer.datasets.**LabeledImageDataset**(*pairs*, *root='.'*, *dtype=None*, *label_dtype=<class 'numpy.int32'>*)
Dataset of image and label pairs built from a list of paths and labels.

This dataset reads an external image file like *ImageDataset*. The difference from *ImageDataset* is that this dataset also returns a label integer. The paths and labels are given as either a list of pairs or a text file contains paths/labels pairs in distinct lines. In the latter case, each path and corresponding label are separated by white spaces. This format is same as one used in Caffe.

Note: **This dataset requires the Pillow package being installed.** In order to use this dataset, install Pillow (e.g. by using the command `pip install Pillow`). Be careful to prepare appropriate libraries for image formats you want to use (e.g. libpng for PNG images, and libjpeg for JPG images).

Warning: You are responsible for preprocessing the images before feeding them to a model. For example, if your dataset contains both RGB and grayscale images, make sure that you convert them to the same format. Otherwise you will get errors because the input dimensions are different for RGB and grayscale images.

Parameters

- **pairs** (*str or list of tuples*) – If it is a string, it is a path to a text file that contains paths to images in distinct lines. If it is a list of pairs, the `i`-th element represents a pair of the path to the `i`-th image and the corresponding label. In both cases, each path is a relative one from the root path given by another argument.
- **root** (*str*) – Root directory to retrieve images from.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).
- **label_dtype** – Data type of the labels.

Methods

__getitem__(*index*)

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

get_example(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.datasets.LabeledZippedImageDataset

class chainer.datasets.**LabeledZippedImageDataset**(*zipfilename*, *labelfilename*, *dtype=None*, *label_dtype=<class 'numpy.int32'>*)

Dataset of zipped image and label pairs.

This dataset is zip version of *LabeledImageDataset*. It takes a zipfile like *ZippedImageDataset*. The label file shall contain lines like text file used in *LabeledImageDataset*, but a filename in each line of the label file shall match with a file in the zip archive.

Parameters

- **zipfilename** (*str*) – Path to a zipfile with images
- **labelfilename** (*str*) – Path to a label file. `i`-th line shall contain a filename and an integer label that corresponds to the `i`-th sample. A filename in the label file shall match with a filename in the zip file given with *zipfilename*.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).
- **label_dtype** – Data type of the labels.

Methods

`__getitem__`(*index*)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__`()
Returns the number of data points.

`get_example`(*i*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

TextDataset

chainer.datasets.TextDataset	Dataset of a line-oriented text file.

chainer.datasets.TextDataset

class chainer.datasets.**TextDataset**(*paths*, *encoding=None*, *errors=None*, *newline=None*, *filter_func=None*)

Dataset of a line-oriented text file.

This dataset reads each line of text file(s) on every call of the *__getitem__()* operator. Positions of line boundaries are cached so that you can quickliy random access the text file by the line number.

Note: Cache will be built in the constructor. You can pickle and unpickle the dataset to reuse the cache, but in that case you are responsible to guarantee that files are not modified after the cache has built.

Parameters

- **paths** (*str or list of str*) – Path to the text file(s). If it is a string, this dataset reads a line from the text file and emits it as str. If it is a list of string, this dataset reads lines from each text file and emits it as a tuple of str. In this case, number of lines in all files must be the same.
- **encoding** (*str or list of str*) – Name of the encoding used to decode the file. See the description in open() for the supported options and how it works. When reading from multiple text files, you can also pass a list of str to use different encoding for each file.
- **errors** (*str or list of str*) – String that specifies how decoding errors are to be handled. See the description in open() for the supported options and how it works. When reading from multiple text files, you can also pass a list of str to use different error handling policy for each file.
- **newline** (*str or list of str*) – Controls how universal newlines mode works. See the description in open() for the supported options and how it works. When reading from multiple text files, you can also pass a list of str to use different mode for each file.
- **filter_func** (*callable*) – Function to filter each line of the text file. It should be a function that takes number of arguments equals to the number of files. Arguments are lines loaded from each file. The filter function must return True to accept the line, or return False to skip the line.

Methods

__getitem__(*index*)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

close()
Manually closes all text files.

In most cases, you do not have to call this method, because files will automatically be closed after TextDataset instance goes out of scope.

get_example(*idx*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

PickleDataset

chainer.datasets.PickleDataset	Dataset stored in a storage using pickle.
chainer.datasets.PickleDatasetWriter	Writer class that makes PickleDataset.
chainer.datasets.open_pickle_dataset	Opens a dataset stored in a given path.
chainer.datasets.open_pickle_dataset_writer	Opens a writer to make a PickleDataset.

chainer.datasets.PickleDataset

class chainer.datasets.**PickleDataset**(*reader*)
Dataset stored in a storage using pickle.

pickle is the default serialization library of Python. This dataset stores any objects in a storage using pickle. Even when a user wants to use a large dataset, this dataset can stores all data in a large storage like HDD and each data can be randomly accessible.

```
>>> with chainer.datasets.open_pickle_dataset_writer(path_to_data) as w:
...     w.write((1, 2.0, 'hello'))
...     w.write((2, 3.0, 'good-bye'))
...
>>> with chainer.datasets.open_pickle_dataset(path_to_data) as dataset:
...     print(dataset[1])
...
(2, 3.0, 'good-bye')
```

Parameters **reader** – File like object. *reader* must support random access.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__getitem__(*index*)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the *get_example()* method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by *get_example*. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by *get_example*.

Example

```
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]   # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

close()
Closes a file reader.

After a user calls this method, the dataset will no longer be accessible..

get_example(*index*)
Returns the i-th example.

Implementations should override it. It should raise `IndexError` if the index is invalid.

Parameters **i** (*int*) – The index of the example.

Returns The i-th example.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.datasets.PickleDatasetWriter

class chainer.datasets.**PickleDatasetWriter**(*writer*, *protocol=4*)
Writer class that makes PickleDataset.

To make *PickleDataset*, a user needs to prepare data using *PickleDatasetWriter*.

Parameters

- **writer** – File like object that supports `write` and `tell` methods.
- **protocol** (*int*) – Valid protocol for `pickle`.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

close()

flush()

write(*x*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.datasets.open_pickle_dataset

chainer.datasets.**open_pickle_dataset**(*path*)
Opens a dataset stored in a given path.

This is a helper function to open *PickleDataset*. It opens a given file in binary mode, and creates a *PickleDataset* instance.

This method does not close the opened file. A user needs to call *PickleDataset.close()* or use *with*:

```
with chainer.datasets.open_pickle_dataset('path') as dataset:
    pass  # use dataset
```

Parameters **path** (*str*) – Path to a dataset.

Returns Opened dataset.

Return type *chainer.datasets.PickleDataset*

chainer.datasets.open_pickle_dataset_writer

chainer.datasets.**open_pickle_dataset_writer**(*path*, *protocol=4*)

Opens a writer to make a PickleDataset.

This is a helper function to open *PickleDatasetWriter*. It opens a given file in binary mode and creates a *PickleDatasetWriter* instance.

This method does not close the opened file. A user needs to call *PickleDatasetWriter.close()* or use *with*:

```
with chainer.datasets.open_pickle_dataset_writer('path') as writer:
    pass  # use writer
```

Parameters

- **path** (*str*) – Path to a dataset.
- **protocol** (*int*) – Valid protocol for pickle.

Returns Opened writer.

Return type *chainer.datasets.PickleDatasetWriter*

4.9.4 Concrete Datasets

chainer.datasets.get_mnist	Gets the MNIST dataset.
chainer.datasets.get_kuzushiji_mnist	Gets the Kuzushiji-MNIST dataset.
chainer.datasets.get_kuzushiji_mnist_labels	Provides a list of labels for the Kuzushiji-MNIST dataset.
chainer.datasets.get_fashion_mnist_labels	Provide a list of the string value names of the labels.
chainer.datasets.get_fashion_mnist	Gets the Fashion-MNIST dataset.
chainer.datasets.get_cifar10	Gets the CIFAR-10 dataset.
chainer.datasets.get_cifar100	Gets the CIFAR-100 dataset.
chainer.datasets.get_ptb_words	Gets the Penn Tree Bank dataset as long word sequences.
chainer.datasets.get_ptb_words_vocabulary	Gets the Penn Tree Bank word vocabulary.
chainer.datasets.get_svhn	Gets the SVHN dataset.

chainer.datasets.get_mnist

chainer.datasets.**get_mnist**(*withlabel=True*, *ndim=1*, *scale=1.0*, *dtype=None*, *label_dtype=<class 'numpy.int32'>*, *rgb_format=False*)

Gets the MNIST dataset.

MNIST is a set of hand-written digits represented by grey-scale 28x28 images. In the original images, each pixel is represented by one-byte unsigned integer. This function scales the pixels to floating point values in the interval `[0, scale]`.

This function returns the training set and the test set of the official MNIST dataset. If `withlabel` is `True`, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- **withlabel** (*bool*) – If `True`, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **ndim** (*int*) – Number of dimensions of each image. The shape of each image is determined depending on `ndim` as follows:
  - `ndim == 1`: the shape is `(784,)`
  - `ndim == 2`: the shape is `(28, 28)`
  - `ndim == 3`: the shape is `(1, 28, 28)`
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval `[0, 1]`.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).
- **label_dtype** – Data type of the labels.
- **rgb_format** (*bool*) – if `ndim == 3` and `rgb_format` is `True`, the image will be converted to rgb format by duplicating the channels so the image shape is (3, 28, 28). Default is `False`.

Returns A tuple of two datasets. If `withlabel` is `True`, both datasets are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_kuzushiji_mnist

`chainer.datasets.get_kuzushiji_mnist`(*withlabel=True*, *ndim=1*, *scale=1.0*, *dtype=None*, *label_dtype=<class 'numpy.int32'>*, *rgb_format=False*)

Gets the Kuzushiji-MNIST dataset.

Kuzushiji-MNIST (KMNIST) is a set of hand-written Japanese characters represented by grey-scale 28x28 images. In the original images, each pixel is represented by one-byte unsigned integer. This function scales the pixels to floating point values in the interval `[0, scale]`.

This function returns the training set and the test set of the official KMNIST dataset. If `withlabel` is `True`, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- **withlabel** (*bool*) – If `True`, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **ndim** (*int*) – Number of dimensions of each image. The shape of each image is determined depending on `ndim` as follows:
  - `ndim == 1`: the shape is `(784,)`
  - `ndim == 2`: the shape is `(28, 28)`
  - `ndim == 3`: the shape is `(1, 28, 28)`
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval `[0, 1]`.

- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).
- **label_dtype** – Data type of the labels.
- **rgb_format** (*bool*) – if `ndim == 3` and `rgb_format` is `True`, the image will be converted to rgb format by duplicating the channels so the image shape is (3, 28, 28). Default is `False`.

Returns A tuple of two datasets. If `withlabel` is `True`, both datasets are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_kuzushiji_mnist_labels

`chainer.datasets.get_kuzushiji_mnist_labels()`

Provides a list of labels for the Kuzushiji-MNIST dataset.

Returns List of labels in the form of tuples. Each tuple contains the character name in romaji as a string value and the unicode codepoint for the character.

chainer.datasets.get_fashion_mnist_labels

`chainer.datasets.get_fashion_mnist_labels()`

Provide a list of the string value names of the labels.

Returns List of string values of the image labels.

chainer.datasets.get_fashion_mnist

`chainer.datasets.get_fashion_mnist(withlabel=True, ndim=1, scale=1.0, dtype=None, label_dtype=<class 'numpy.int32'>, rgb_format=False)`

Gets the Fashion-MNIST dataset.

Fashion-MNIST is a set of fashion articles represented by grey-scale 28x28 images. In the original images, each pixel is represented by one-byte unsigned integer. This function scales the pixels to floating point values in the interval `[0, scale]`.

This function returns the training set and the test set of the official Fashion-MNIST dataset. If `withlabel` is `True`, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- **withlabel** (*bool*) – If `True`, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **ndim** (*int*) – Number of dimensions of each image. The shape of each image is determined depending on `ndim` as follows:
  - `ndim == 1`: the shape is `(784,)`
  - `ndim == 2`: the shape is `(28, 28)`
  - `ndim == 3`: the shape is `(1, 28, 28)`
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval `[0, 1]`.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).

- **label_dtype** – Data type of the labels.
- **rgb_format** (*bool*) – if `ndim == 3` and `rgb_format` is `True`, the image will be converted to rgb format by duplicating the channels so the image shape is (3, 28, 28). Default is `False`.

Returns A tuple of two datasets. If `withlabel` is `True`, both datasets are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_cifar10

`chainer.datasets.get_cifar10`(*withlabel=True*, *ndim=3*, *scale=1.0*, *dtype=None*)

Gets the CIFAR-10 dataset.

CIFAR-10 is a set of small natural images. Each example is an RGB color image of size 32x32, classified into 10 groups. In the original images, each component of pixels is represented by one-byte unsigned integer. This function scales the components to floating point values in the interval `[0, scale]`.

This function returns the training set and the test set of the official CIFAR-10 dataset. If `withlabel` is `True`, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- **withlabel** (*bool*) – If `True`, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **ndim** (*int*) – Number of dimensions of each image. The shape of each image is determined depending on ndim as follows:
  - `ndim == 1`: the shape is `(3072,)`
  - `ndim == 3`: the shape is `(3, 32, 32)`
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval `[0, 1]`.
- **dtype** – Data type of resulting image arrays. `chainer.config.dtype` is used by default (see *Configuring Chainer*).

Returns A tuple of two datasets. If `withlabel` is `True`, both datasets are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_cifar100

`chainer.datasets.get_cifar100`(*withlabel=True*, *ndim=3*, *scale=1.0*, *dtype=None*)

Gets the CIFAR-100 dataset.

CIFAR-100 is a set of small natural images. Each example is an RGB color image of size 32x32, classified into 100 groups. In the original images, each component pixels is represented by one-byte unsigned integer. This function scales the components to floating point values in the interval `[0, scale]`.

This function returns the training set and the test set of the official CIFAR-100 dataset. If `withlabel` is `True`, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- **withlabel** (*bool*) – If `True`, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **ndim** (*int*) – Number of dimensions of each image. The shape of each image is determined depending on ndim as follows:

  - ndim == 1: the shape is (3072,)
  - ndim == 3: the shape is (3, 32, 32)
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval [0, 1].
- **dtype** – Data type of resulting image arrays. chainer.config.dtype is used by default (see *Configuring Chainer*).

Returns A tuple of two datasets. If withlabel is True, both are *TupleDataset* instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_ptb_words

chainer.datasets.**get_ptb_words**()

Gets the Penn Tree Bank dataset as long word sequences.

Penn Tree Bank is originally a corpus of English sentences with linguistic structure annotations. This function uses a variant distributed at https://github.com/wojzaremba/lstm, which omits the annotation and splits the dataset into three parts: training, validation, and test.

This function returns the training, validation, and test sets, each of which is represented as a long array of word IDs. All sentences in the dataset are concatenated by End-of-Sentence mark '<eos>', which is treated as one of the vocabulary.

Returns Int32 vectors of word IDs.

Return type tuple of numpy.ndarray

See also:

Use *get_ptb_words_vocabulary()* to get the mapping between the words and word IDs.

chainer.datasets.get_ptb_words_vocabulary

chainer.datasets.**get_ptb_words_vocabulary**()

Gets the Penn Tree Bank word vocabulary.

Returns Dictionary that maps words to corresponding word IDs. The IDs are used in the Penn Tree Bank long sequence datasets.

Return type dict

See also:

See *get_ptb_words()* for the actual datasets.

chainer.datasets.get_svhn

chainer.datasets.**get_svhn**(*withlabel=True*, *scale=1.0*, *dtype=None*, *label_dtype=<class 'numpy.int32'>*, *add_extra=False*)

Gets the SVHN dataset.

The Street View House Numbers (SVHN) dataset is a dataset similar to MNIST but composed of cropped images of house numbers. The functionality of this function is identical to the counterpart for the MNIST dataset (*get_mnist()*), with the exception that there is no ndim argument.

Note: SciPy is required to use this feature.

Parameters

- **withlabel** (*bool*) – If True, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- **scale** (*float*) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval [0, 1].
- **dtype** – Data type of resulting image arrays. chainer.config.dtype is used by default (see *Configuring Chainer*).
- **label_dtype** – Data type of the labels.
- **add_extra** – Use extra training set.

Returns If add_extra is False, a tuple of two datasets (train and test). Otherwise, a tuple of three datasets (train, test, and extra). If withlabel is True, all datasets are *TupleDataset* instances. Otherwise, both datasets are arrays of images.

Note: ChainerCV supports implementations of datasets that are useful for computer vision problems, which can be found in chainercv.datasets. Here is a subset of data loaders supported by ChainerCV:

- **Bounding Box Datasets**
  - chainercv.datasets.VOCBboxDataset
  - chainercv.datasets.COCOBboxDataset
- **Semantic Segmentation Datasets**
  - chainercv.datasets.ADE20KSemanticSegmentationDataset
  - chainercv.datasets.CamVidDataset
  - chainercv.datasets.CityscapesSemanticSegmentationDataset
  - chainercv.datasets.VOCSemanticSegmentationDataset
- **Instance Segmentation Datasets**
  - chainercv.datasets.COCOInstanceSegmentationDataset
  - chainercv.datasets.VOCInstanceSegmentationDataset
- **Classification Datasets**
  - chainercv.datasets.CUBLabelDataset
  - chainercv.datasets.OnlineProductsDataset

4.10 Iterator

Chainer provides some iterators that implement typical strategies to create mini-batches by iterating over datasets. *SerialIterator* is the simplest one, which extracts mini-batches in the main thread. *MultiprocessIterator* and *MultithreadIterator* are parallelized versions of *SerialIterator*. They maintain worker subprocesses and subthreads, respectively, to load the next mini-batch in parallel.

chainer.iterators.SerialIterator	Dataset iterator that serially reads the examples.
chainer.iterators.MultiprocessIterator	Dataset iterator that loads examples in parallel.
chainer.iterators.MultithreadIterator	Dataset iterator that loads examples in parallel.
chainer.iterators.DaliIterator	(Experimental) Iterator for DALI pipeline.

4.10.1 chainer.iterators.SerialIterator

class chainer.iterators.**SerialIterator**(*dataset*, *batch_size*, *repeat=True*, *shuffle=None*, *order_sampler=None*)

Dataset iterator that serially reads the examples.

This is a simple implementation of *Iterator* that just visits each example in either the order of indexes or a shuffled order.

To avoid unintentional performance degradation, the `shuffle` option is set to `True` by default. For validation, it is better to set it to `False` when the underlying dataset supports fast slicing. If the order of examples has an important meaning and the updater depends on the original order, this option should be set to `False`.

This iterator saves `-1` instead of `None` in snapshots since some serializers do not support `None`.

Parameters

- **dataset** – Dataset to iterate.
- **batch_size** (*int*) – Number of examples within each batch.
- **repeat** (*bool*) – If `True`, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.
- **shuffle** (*bool*) – If `True`, the order of examples is shuffled at the beginning of each epoch. Otherwise, examples are extracted in the order of indexes. If `None` and no `order_sampler` is given, the behavior is the same as the case with `shuffle=True`.
- **order_sampler** (*callable*) – A callable that generates the order of the indices to sample in the next epoch when a epoch finishes. This function should take two arguments: the current order and the current position of the iterator. This should return the next order. The size of the order should remain constant. This option cannot be used when `shuffle` is not `None`.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__next__()

Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

__iter__()

Returns self.

finalize()

Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

This method can be called multiple times.

next()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

reset()

serialize(*serializer*)
Serializes the internal state of the iterator.

This is a method to support the serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serialize what is set manually by users such as the batch size.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

current_position

epoch

epoch_detail

is_new_epoch

previous_epoch_detail

repeat

4.10.2 chainer.iterators.MultiprocessIterator

class chainer.iterators.**MultiprocessIterator**(*dataset*, *batch_size*, *repeat=True*, *shuffle=None*, *n_processes=None*, *n_prefetch=1*, *shared_mem=None*, *order_sampler=None*, *dataset_timeout=30.0*, *maxtasksperchild=None*)

Dataset iterator that loads examples in parallel.

This is an implementation of *Iterator* that loads examples with worker processes. It uses the standard `multiprocessing` module to parallelize the loading. The dataset is sent to the worker processes in the standard way using pickle.

Note that this iterator effectively prefetches the examples for the next batch asynchronously after the current batch is returned.

This iterator saves `-1` instead of `None` in snapshots since some serializers do not support `None`.

Note: When you are using OpenCV somewhere in your code and the `MultiprocessIterator` is used in the training code, the training loop may get stuck at some point. In such situation, there are several workarounds to prevent the process got stuck.

1. Set the environment variable as follows: `OMP_NUM_THREADS=1`
2. Add `cv2.setNumThreads(0)` right after `import cv2` in your training script.
3. Use *MultithreadIterator* instead of `MultiprocessIterator`.

Parameters

- **dataset** (*Dataset*) – Dataset to iterate.
- **batch_size** (*int*) – Number of examples within each batch.
- **repeat** (*bool*) – If `True`, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.
- **shuffle** (*bool*) – If `True`, the order of examples is shuffled at the beginning of each epoch. Otherwise, examples are extracted in the order of indexes. If `None` and no `order_sampler` is given, the behavior is the same as the case with `shuffle=True`.
- **n_processes** (*int*) – Number of worker processes. The number of CPUs is used by default.
- **n_prefetch** (*int*) – Number of prefetch batches.
- **shared_mem** (*int*) – The size of using shared memory per data. If `None`, size is adjusted automatically.
- **dataset_timeout** (*float*) – `MultiprocessIterator.TimeoutWarning` will be issued after this time in seconds elapsed in each dataset realization. `None` to disable the warning. You can turn this warning into an error by using `warnings.simplefilter()`:

```
warnings.simplefilter(
    'error',
    chainer.iterators.MultiprocessIterator.TimeoutWarning)
```

- **order_sampler** (*callable*) – A callable that generates the order of the indices to sample in the next epoch when a epoch finishes. This function should take two arguments:

the current order and the current position of the iterator. This should return the next order. The size of the order should remain constant. This option cannot be used when `shuffle` is not `None`.

- **maxtasksperchild** (*int*) – Number of tasks a worker of prefetch process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. If `None`, worker processes will live as long as the pool.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__next__()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

__iter__()
Returns self.

__copy__()

finalize()
Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

This method can be called multiple times.

next()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

reset()

serialize(*serializer*)
Serializes the internal state of the iterator.

This is a method to support the serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serialize what is set manually by users such as the batch size.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

current_position

epoch

epoch_detail

is_new_epoch

previous_epoch_detail

4.10.3 chainer.iterators.MultithreadIterator

class chainer.iterators.**MultithreadIterator**(*dataset*, *batch_size*, *repeat=True*, *shuffle=None*, *n_threads=1*, *order_sampler=None*)

Dataset iterator that loads examples in parallel.

This is an implementation of *Iterator* that loads examples with worker threads. It uses the standard threading module to parallelize the loading.

Note that this iterator effectively prefetches the examples for the next batch asynchronously after the current batch is returned.

This iterator saves -1 instead of None in snapshots since some serializers do not support None.

Parameters

- **dataset** (*Dataset*) – Dataset to iterate.
- **batch_size** (*int*) – Number of examples within each batch.
- **repeat** (*bool*) – If True, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.
- **shuffle** (*bool*) – If True, the order of examples is shuffled at the beginning of each epoch. Otherwise, examples are extracted in the order of indexes. If None and no order_sampler is given, the behavior is the same as the case with shuffle=True.
- **n_threads** (*int*) – Number of worker threads.
- **order_sampler** (*callable*) – A callable that generates the order of the indices to sample in the next epoch when a epoch finishes. This function should take two arguments: the current order and the current position of the iterator. This should return the next order. The size of the order should remain constant. This option cannot be used when shuffle is not None.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__next__()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the StopIteration exception when it stops the iteration.

__iter__()
Returns self.

finalize()
Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

This method can be called multiple times.

next()
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the StopIteration exception when it stops the iteration.

reset()

serialize(*serializer*)
Serializes the internal state of the iterator.

This is a method to support the serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serialize what is set manually by users such as the batch size.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

current_position

epoch

epoch_detail

is_new_epoch

previous_epoch_detail

repeat

4.10.4 chainer.iterators.DaliIterator

class chainer.iterators.**DaliIterator**(*pipeline*, *repeat=True*)

(Experimental) Iterator for DALI pipeline.

Parameters

- **pipeline** – DALI pipeline.
- **repeat** (*bool*) – If `True`, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.

Methods

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__next__()

Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

__iter__()

Returns self.

finalize()

Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

This method can be called multiple times.

next()

Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the `StopIteration` exception when it stops the iteration.

reset()

serialize(*serializer*)

Serializes the internal state of the iterator.

This is a method to support the serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serialize what is set manually by users such as the batch size.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

batch_size

epoch_detail

previous_epoch_detail

repeat

4.10.5 Order sampler examples

An Iterator iterates over a dataset according to an order represented by a 1-D array of indices. Order samplers are callables that are used by those iterators to generate this array.

chainer.iterators.OrderSampler	Base class of all order samplers.
chainer.iterators.ShuffleOrderSampler	Sampler that generates random orders.

chainer.iterators.OrderSampler

class chainer.iterators.**OrderSampler**
Base class of all order samplers.

Every order sampler subclass has to provide a method *__call__()*. This method is called by an iterator before a new epoch, and it should return a new index order for the next epoch.

Methods

__call__(*current_order*, *current_position*)
Sample the next order.

Parameters

- **current_order** (*numpy.ndarray*) – 1-D array of indices. The length should be the same as the dataset to sample data from.
- **current_position** (*int*) – The current position of an iterator.

Returns 1-D array of indices. This is the order in which examples are sampled from a dataset in the next epoch.

Return type numpy.ndarray

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.iterators.ShuffleOrderSampler

class chainer.iterators.**ShuffleOrderSampler**(*random_state=None*)
Sampler that generates random orders.

This is expected to be used together with Chainer's iterators. An order sampler is called by an iterator every epoch.

The two initializations below create basically the same objects.

```
>>> dataset = [(1, 2), (3, 4)]
>>> it = chainer.iterators.MultiprocessIterator(dataset, 1, shuffle=True)
>>> it = chainer.iterators.MultiprocessIterator(
...     dataset, 1, order_sampler=chainer.iterators.ShuffleOrderSampler())
```

Parameters **random_state** (*numpy.random.RandomState*) – Pseudo-random number generator.

Methods

__call__(*current_order*, *current_position*)
Sample the next order.

Parameters

- **current_order** (*numpy.ndarray*) – 1-D array of indices. The length should be the same as the dataset to sample data from.
- **current_position** (*int*) – The current position of an iterator.

Returns 1-D array of indices. This is the order in which examples are sampled from a dataset in the next epoch.

Return type numpy.ndarray

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.11 Serializers

4.11.1 Serialization in NumPy NPZ format

NumPy serializers can be used in arbitrary environments that Chainer runs with. It consists of asymmetric serializer/deserializer due to the fact that `numpy.savez()` does not support online serialization. Therefore, serialization requires two-step manipulation: first packing the objects into a flat dictionary, and then serializing it into npz format.

chainer.serializers.DictionarySerializer	Serializer for dictionary.
chainer.serializers.NpzDeserializer	Deserializer for NPZ format.
chainer.serializers.save_npz	Saves an object to the file in NPZ format.
chainer.serializers.load_npz	Loads an object from the file in NPZ format.

chainer.serializers.DictionarySerializer

class chainer.serializers.**DictionarySerializer**(*target=None*, *path=''*)
Serializer for dictionary.

This is the standard serializer in Chainer. The hierarchy of objects are simply mapped to a flat dictionary with keys representing the paths to objects in the hierarchy.

Note: Despite of its name, this serializer DOES NOT serialize the object into external files. It just build a flat dictionary of arrays that can be fed into `numpy.savez()` and `numpy.savez_compressed()`. If you want to use this serializer directly, you have to manually send a resulting dictionary to one of these functions.

Parameters

- **target** (*dict*) – The dictionary that this serializer saves the objects to. If target is None, then a new dictionary is created.
- **path** (*str*) – The base path in the hierarchy that this serializer indicates.

Variables **target** (*dict*) – The target dictionary. Once the serialization completes, this dictionary can be fed into `numpy.savez()` or `numpy.savez_compressed()` to serialize it in the NPZ format.

Methods

__call__(*key*, *value*)

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

__getitem__(*key*)

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters **key** (*str*) – Name of the child serializer.

save(*obj*)

Saves an object by this serializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

chainer.serializers.NpzDeserializer

class chainer.serializers.**NpzDeserializer**(*npz*, *path=''*, *strict=True*, *ignore_names=None*)

Deserializer for NPZ format.

This is the standard deserializer in Chainer. This deserializer can be used to read an object serialized by `save_npz()`.

Parameters

- **npz** – *npz* file object.
- **path** – The base path that the deserialization starts from.
- **strict** (*bool*) – If `True`, the deserializer raises an error when an expected value is not found in the given NPZ file. Otherwise, it ignores the value and skip deserialization.
- **ignore_names** (*string, callable or list of them*) – If callable, it is a function that takes a name of a parameter and a persistent and returns `True` when it needs to be skipped. If string, this is a name of a parameter or persistent that are going to be skipped. This can also be a list of callables and strings that behave as described above.

Methods

__call__(*key*, *value*)

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

__getitem__(*key*)

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters **key** (*str*) – Name of the child serializer.

load(*obj*)

Loads an object from this deserializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()

Return self<value.

__le__()

Return self<=value.

__gt__()

Return self>value.

__ge__()

Return self>=value.

chainer.serializers.save_npz

`chainer.serializers.`**save_npz**(*file*, *obj*, *compression=True*)

Saves an object to the file in NPZ format.

This is a short-cut function to save only one object into an NPZ file.

Parameters

- **file** (*str or file-like*) – Target file to write to.
- **obj** – Object to be serialized. It must support serialization protocol. If it is a dictionary object, the serialization will be skipped.
- **compression** (*bool*) – If `True`, compression in the resulting zip file is enabled.

See also:

chainer.serializers.load_npz()

chainer.serializers.load_npz

`chainer.serializers.`**load_npz**(*file*, *obj*, *path=''*, *strict=True*, *ignore_names=None*)

Loads an object from the file in NPZ format.

This is a short-cut function to load from an *.npz* file that contains only one object.

Parameters

- **file** (*str or file-like*) – File to be loaded.
- **obj** – Object to be deserialized. It must support serialization protocol.
- **path** (*str*) – The path in the hierarchy of the serialized data under which the data is to be loaded. The default behavior (blank) will load all data under the root path.
- **strict** (*bool*) – If `True`, the deserializer raises an error when an expected value is not found in the given NPZ file. Otherwise, it ignores the value and skip deserialization.

- **ignore_names** (*string, callable or list of them*) – If callable, it is a function that takes a name of a parameter and a persistent and returns `True` when it needs to be skipped. If string, this is a name of a parameter or persistent that are going to be skipped. This can also be a list of callables and strings that behave as described above.

See also:

chainer.serializers.save_npz()

4.11.2 Serialization in HDF5 format

chainer.serializers.HDF5Serializer	Serializer for HDF5 format.
chainer.serializers.HDF5Deserializer	Deserializer for HDF5 format.
chainer.serializers.save_hdf5	Saves an object to the file in HDF5 format.
chainer.serializers.load_hdf5	Loads an object from the file in HDF5 format.

chainer.serializers.HDF5Serializer

class chainer.serializers.**HDF5Serializer**(*group*, *compression=4*)

Serializer for HDF5 format.

This is the standard serializer in Chainer. The chain hierarchy is simply mapped to HDF5 hierarchical groups.

Parameters

- **group** (*h5py.Group*) – The group that this serializer represents.
- **compression** (*int*) – Gzip compression level.

Methods

__call__(*key*, *value*)

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

`__getitem__`(*key*)
Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters **key** (*str*) – Name of the child serializer.

`save`(*obj*)
Saves an object by this serializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

`__eq__`()
Return self==value.

`__ne__`()
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

`__ge__`()
Return self>=value.

chainer.serializers.HDF5Deserializer

class `chainer.serializers.`**`HDF5Deserializer`**(*group*, *strict=True*)
Deserializer for HDF5 format.

This is the standard deserializer in Chainer. This deserializer can be used to read an object serialized by *HDF5Serializer*.

Parameters

- **group** (*h5py.Group*) – The group that the deserialization starts from.
- **strict** (*bool*) – If `True`, the deserializer raises an error when an expected value is not found in the given HDF5 file. Otherwise, it ignores the value and skip deserialization.

Methods

`__call__`(*key*, *value*)
Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

__getitem__(*key*)
Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters key (*str*) – Name of the child serializer.

load(*obj*)
Loads an object from this deserializer.

This is equivalent to `obj.serialize(self)`.

Parameters obj – Target object to be serialized.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.serializers.save_hdf5

`chainer.serializers.`**save_hdf5**(*filename, obj, compression=4*)
Saves an object to the file in HDF5 format.

This is a short-cut function to save only one object into an HDF5 file. If you want to save multiple objects to one HDF5 file, use *HDF5Serializer* directly by passing appropriate `h5py.Group` objects.

Parameters

- **filename** (*str*) – Target file name.

- **obj** – Object to be serialized. It must support serialization protocol. If it is a dictionary object, the serialization will be skipped.
- **compression** (*int*) – Gzip compression level.

Note: Currently `save_hdf5()` only supports writing to an actual file on file system due to a limitation of HD5F library. See h5py/h5py#687 for details.

See also:

`chainer.serializers.load_hdf5()`

chainer.serializers.load_hdf5

`chainer.serializers.load_hdf5(filename, obj)`

Loads an object from the file in HDF5 format.

This is a short-cut function to load from an HDF5 file that contains only one object. If you want to load multiple objects from one HDF5 file, use `HDF5Deserializer` directly by passing appropriate `h5py.Group` objects.

Parameters

- **filename** (*str*) – Name of the file to be loaded.
- **obj** – Object to be deserialized. It must support serialization protocol.

Note: Currently `load_hdf5()` only supports loading an actual file on file system due to a limitation of HD5F library. See h5py/h5py#687 for details.

See also:

`chainer.serializers.save_hdf5()`

4.11.3 Serializers base classes

`chainer.Serializer`	Base class of all serializers.
`chainer.AbstractSerializer`	Abstract base class of all serializers and deserializers.
`chainer.Deserializer`	Base class of all deserializers.

chainer.Serializer

class `chainer.Serializer`

Base class of all serializers.

Methods

`__call__(key, value)`

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

__getitem__(*key*)
Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters **key** (*str*) – Name of the child serializer.

save(*obj*)
Saves an object by this serializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.AbstractSerializer

class chainer.**AbstractSerializer**
Abstract base class of all serializers and deserializers.

Methods

__call__(*key*, *value*)
Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

__getitem__(*key*)
Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters key (*str*) – Name of the child serializer.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.Deserializer

class chainer.**Deserializer**
Base class of all deserializers.

Methods

`__call__`(*key*, *value*)

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the `value` argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the `value` argument. String values are treated like scalars.

Note: Serializers and deserializers are required to correctly handle the `None` value. When `value` is `None`, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved `None` value is loaded by a deserializer, it should quietly return the `None` value without modifying the `value` object.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, numpy.ndarray, cupy.ndarray, None, or str*) – Object to be (de)serialized. `None` is only supported by deserializers.

Returns Serialized or deserialized value.

`__getitem__`(*key*)

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters **key** (*str*) – Name of the child serializer.

`load`(*obj*)

Loads an object from this deserializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

`__eq__`()

Return self==value.

`__ne__`()

Return self!=value.

`__lt__`()

Return self<value.

`__le__`()

Return self<=value.

`__gt__`()

Return self>value.

`__ge__`()

Return self>=value.

4.12 Backends and Devices

4.12.1 Common Classes and Utilities

chainer.backend.Device	A base class of unified devices.
chainer.get_device	Returns a device object.
chainer.using_device	Context manager to apply the thread-local device state.
chainer.backend.get_device_from_array	Gets the device from arrays.
chainer.backend.get_array_module	Gets an appropriate NumPy-compatible module to process arguments
chainer.DeviceResident	A base class of objects with multi-device hierarchy.
chainer.device_resident.DeviceResidentsVisitor	Base class of visitors that visits device resident objects recursively.
chainer.backend.copyto	Copies the elements of an ndarray to those of another one.

chainer.backend.Device

class chainer.backend.**Device**

A base class of unified devices.

Chainer has the following concrete implementations:

- *chainer.backend.CpuDevice*
- *chainer.backend.GpuDevice*
- *chainer.backend.Intel64Device*
- *chainer.backend.ChainerxDevice*

Methods

__enter__()

A dummy definition that simply raises RuntimeError.

chainer.using_device() should be used instead.

__exit__(*exc_type*, *exc_value*, *traceback*)

A dummy definition that should never be called.

create_context()

Returns a context manager in which the device is made current.

See also:

chainer.using_device() calls this method internally.

is_array_supported(*array*)

Returns if the specified array is compatible with the device. :param array: An array to be checked :type array: *N-dimensional array*

Returns True if the array is compatible with the device. Otherwise False is returned.

send(*arrays*)

Transfers given arrays to the device.

Parameters **arrays** – Array or arrays of NumPy, CuPy, or ChainerX.

Returns Transferred arrays.

use()
Makes the device current in the current thread.

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name
A unique name of the device.

supported_array_types
Array types supported by the device.

Returns tuple of array types which the device's module functions can handle.

xp
Array module corresponding to the device.

chainer.get_device

chainer.**get_device**(*device_spec*)
Returns a device object.

Parameters **device_spec** (*object*) – Device specifier. If a *chainer.backend.Device* instance is given, it is returned intact. Otherwise the following values are supported:

- ChainerX devices
  - A string representing a device. (ex. `'native:0'`, `'native'`)
  - A `chainerx.Device` object.
- CuPy
  - A string starts with `'@cupy:'`. (ex. `'@cupy:0'`)
  - A `cupy.cuda.Device` object.
- NumPy
  - The string `'@numpy'`.
- NumPy with Intel Architecture

– The string '@intel64'.

chainer.using_device

chainer.**using_device**(*device_spec*)

Context manager to apply the thread-local device state.

Parameters **device_spec** (*object*) – Device specifier. See *chainer.get_device()* for details.

Example

```
with chainer.using_device('@cupy:1'):
    a = cupy.empty((3, 2))

assert a.device.id == 1
```

chainer.backend.get_device_from_array

chainer.backend.**get_device_from_array**(**arrays*)

Gets the device from arrays.

The device on which the given array reside is returned.

Note: Unlike *get_array_module()*, this method does not recognize *Variable* objects. If you need to get device from the *Variable* instance v, you need to use get_device_from_array(v.array).

Parameters **arrays** (*array or list of arrays*) – Arrays to determine the device. If multiple arrays are given, the device correspoinding to the first array which is not NumPy array is returned.

Returns Device instance.

Return type *chainer.backend.Device*

chainer.backend.get_array_module

chainer.backend.**get_array_module**(**args*)

Gets an appropriate NumPy-compatible module to process arguments

This function will return their data arrays' array module for *Variable* arguments.

Parameters **args** – Values to determine whether NumPy, CuPy, or ChainerX should be used.

Returns numpy, cupy, or *chainerx* is returned based on the types of the arguments.

Return type module

chainer.DeviceResident

class chainer.**DeviceResident**

A base class of objects with multi-device hierarchy.

Methods

device_resident_accept(*visitor*)
Applies the visitor to all the device objects in this instance.

Parameters **visitor** (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

from_chx()
Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

to_chx()
Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override *device_resident_accept()* to do so.

Returns: self

to_device(*device*)
Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See *get_device()* for available values.

Returns: self

to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override *device_resident_accept()* to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()
Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use *to_device()* instead.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
Device instance.

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or *chainerx*.

chainer.device_resident.DeviceResidentsVisitor

class chainer.device_resident.**DeviceResidentsVisitor**
Base class of visitors that visits device resident objects recursively.

See also:

chainer.DeviceResident

Methods

visit_array(*arr*)
Processes an array and returns a new one.

If the visitor does not create a new array, it can simply return the original array.

visit_device_resident(*device_resident*)
Processes a *DeviceResident* instance.

visit_variable(*param*)
Processes a *Variable* or a *Parameter*.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.backend.copyto

chainer.backend.**copyto**(*dst*, *src*)
Copies the elements of an ndarray to those of another one.

This function can copy the CPU/GPU arrays to the destination arrays on another device.

Parameters

- **dst** (numpy.ndarray, cupy.ndarray, ideep4py.mdarray or *chainerx.ndarray*) – Destination array.
- **src** (numpy.ndarray, cupy.ndarray, ideep4py.mdarray or *chainerx.ndarray*) – Source array.

4.12.2 Concrete Device Classes

chainer.backend.CpuDevice	Device for CPU (NumPy) backend
chainer.backend.GpuDevice	Device for GPU (CuPy) backend
chainer.backend.Intel64Device	Device for Intel64 (Intel Architecture) backend with iDeep
chainer.backend.ChainerxDevice	Device for ChainerX backend

chainer.backend.CpuDevice

class chainer.backend.**CpuDevice**
Device for CPU (NumPy) backend

Methods

__enter__()
A dummy definition that simply raises RuntimeError.

chainer.using_device() should be used instead.

__exit__(*exc_type*, *exc_value*, *traceback*)
A dummy definition that should never be called.

create_context()
Returns a context manager in which the device is made current.

See also:

chainer.using_device() calls this method internally.

static from_array(*array*)

is_array_supported(*array*)
Returns if the specified array is compatible with the device. :param array: An array to be checked :type array: *N-dimensional array*

Returns `True` if the array is compatible with the device. Otherwise `False` is returned.

send(*arrays*)
Transfers given arrays to the device.

Parameters **arrays** – Array or arrays of NumPy, CuPy, or ChainerX.

Returns Transferred arrays.

send_array(*array*)

use()
Makes the device current in the current thread.

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = '@numpy'

supported_array_types = (<class 'numpy.ndarray'>,)

chainer.backend.GpuDevice

class chainer.backend.**GpuDevice**(*device*)
Device for GPU (CuPy) backend

Methods

__enter__()
A dummy definition that simply raises RuntimeError.

chainer.using_device() should be used instead.

__exit__(*exc_type*, *exc_value*, *traceback*)
A dummy definition that should never be called.

create_context()
Returns a context manager in which the device is made current.

See also:

chainer.using_device() calls this method internally.

static from_array(*array*)

static from_device_id(*device_id*)
Returns a *GpuDevice* corresponding to the CUDA device ID.

is_array_supported(*array*)
Returns if the specified array is compatible with the device. :param array: An array to be checked :type array: *N-dimensional array*

Returns `True` if the array is compatible with the device. Otherwise `False` is returned.

send(*arrays*)
Transfers given arrays to the device.

Parameters **arrays** – Array or arrays of NumPy, CuPy, or ChainerX.

Returns Transferred arrays.

send_array(*array*)

use()
Makes the device current in the current thread.

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name
A unique name of the device.

supported_array_types = (<class 'chainer.backends.cuda.ndarray'>,)

xp = <object object>

chainer.backend.Intel64Device

class chainer.backend.**Intel64Device**
Device for Intel64 (Intel Architecture) backend with iDeep

Methods

__enter__()
A dummy definition that simply raises RuntimeError.

chainer.using_device() should be used instead.

__exit__(*exc_type*, *exc_value*, *traceback*)
A dummy definition that should never be called.

create_context()
Returns a context manager in which the device is made current.

See also:

chainer.using_device() calls this method internally.

static from_array(*array*)

is_array_supported(*array*)
Returns if the specified array is compatible with the device. :param array: An array to be checked :type array: *N-dimensional array*

Returns True if the array is compatible with the device. Otherwise False is returned.

send(*arrays*)
Transfers given arrays to the device.

Parameters arrays – Array or arrays of NumPy, CuPy, or ChainerX.

Returns Transferred arrays.

send_array(*array*)

use()
Makes the device current in the current thread.

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

name = '@intel64'

supported_array_types = (<class 'numpy.ndarray'>, <class 'chainer.backends.intel64.mda

chainer.backend.ChainerxDevice

class `chainer.backend.`**`ChainerxDevice`**(*device*)
Device for ChainerX backend

Methods

`__enter__`()
A dummy definition that simply raises RuntimeError.

`chainer.using_device()` should be used instead.

`__exit__`(*exc_type*, *exc_value*, *traceback*)
A dummy definition that should never be called.

`create_context`()
Returns a context manager in which the device is made current.

See also:

`chainer.using_device()` calls this method internally.

static `from_array`(*array*)

static `from_fallback_device`(*device*)
Returns a *`ChainerxDevice`* corresponding to the fallback device.

See also:

`fallback_device`

`is_array_supported`(*array*)
Returns if the specified array is compatible with the device. :param array: An array to be checked :type array: *N-dimensional array*

Returns `True` if the array is compatible with the device. Otherwise `False` is returned.

`send`(*arrays*)
Transfers given arrays to the device.

Parameters **`arrays`** – Array or arrays of NumPy, CuPy, or ChainerX.

Returns Transferred arrays.

`send_array`(*array*)

`use`()
Makes the device current in the current thread.

`__eq__`(*other*)
Return self==value.

`__ne__`(*other*)
Return self!=value.

`__lt__`()
Return self<value.

`__le__`()
Return self<=value.

`__gt__`()
Return self>value.

__ge__()
Return self>=value.

Attributes

fallback_device
Fallback device.

A fallback device is either a *CpuDevice* or a *GpuDevice* which shares the same physical device with the original ChainerX device.

For example, the fallback device of `native:0` ChainerX device is *CpuDevice*. The fallback device of `cuda:1` ChainerX device is *GpuDevice* with device ID 1.

name
A unique name of the device.

supported_array_types = (<class 'chainerx.ndarray'>,)

4.12.3 GPU (CuPy)

Device, context and memory management on CuPy.

Note: The package `chainer.cuda` has been renamed to *chainer.backends.cuda* as of v4.0.0, but the previous module path `chainer.cuda` is also available.

Chainer uses CuPy (with very thin wrapper) to exploit the speed of GPU computation. Following modules and classes defined in CuPy are imported to *chainer.backends.cuda* module for convenience (refer to this table when reading chainer's source codes).

imported name	original name
chainer.backends.cuda.cupy	cupy
chainer.backends.cuda.cupyx	cupyx
chainer.backends.cuda.ndarray	cupy.ndarray
chainer.backends.cuda.cupy.cuda	cupy.cuda
chainer.backends.cuda.Device	cupy.cuda.Device
chainer.backends.cuda.Event	cupy.cuda.Event
chainer.backends.cuda.Stream	cupy.cuda.Stream

Chainer replaces the default allocator of CuPy by its memory pool implementation. It enables us to reuse the device memory over multiple forward/backward computations, and temporary arrays for consecutive elementwise operations.

Devices

chainer.backends.cuda.get_device	Gets the device from a device object, an ID integer or an array object.
chainer.backends.cuda.get_device_from_id	Gets the device from an ID integer.
chainer.backends.cuda.get_device_from_array	Gets the device from a list of CuPy array or a single CuPy array.

chainer.backends.cuda.get_device

`chainer.backends.cuda.get_device(*args)`

Gets the device from a device object, an ID integer or an array object.

Note: This API is deprecated since v3.0.0. Please use *get_device_from_id()* or *get_device_from_array()* instead.

This is a convenient utility to select a correct device if the type of `arg` is unknown (i.e., one can use this function on arrays that may be on CPU or GPU). The returned device object supports the context management protocol of Python for the *with* statement.

Parameters **args** – Values to specify a GPU device. The first device object, integer or `cupy.ndarray` object is used to select a device. If it is a device object, it is returned. If it is an integer, the corresponding device is returned. If it is a CuPy array, the device on which this array reside is returned. If any arguments are neither integers nor CuPy arrays, a dummy device object representing CPU is returned.

Returns Device object specified by given `args`.

See also:

See `cupy.cuda.Device` for the device selection not by arrays.

chainer.backends.cuda.get_device_from_id

`chainer.backends.cuda.get_device_from_id(device_id)`

Gets the device from an ID integer.

Parameters **device_id** (*int or None*) – The ID of the device which this function returns.

chainer.backends.cuda.get_device_from_array

`chainer.backends.cuda.get_device_from_array(*arrays)`

Gets the device from a list of CuPy array or a single CuPy array.

Deprecated since version v6.0.0: This API is deprecated. Please use *chainer.backend.get_device_from_array()* instead.

The device on which the given CuPy array reside is returned.

Note: This method only recognizes `cupy.ndarray`s in arguments. Especially note that, unlike *get_array_module()*, this method does not recognize *Variable* objects. If you need to get device from the *Variable* instance `v`, you need to use `get_device_from_array(v.array)`.

Parameters **arrays** (`cupy.ndarray` or list of `cupy.ndarray`) – A CuPy array which this function returns the device corresponding to. If a list of `cupy.ndarray`s are given, it returns the first device object of an array in the list.

CuPy array allocation and copy

`chainer.backends.cuda.copy`	Copies a `cupy.ndarray` object using the default stream.
`chainer.backends.cuda.to_cpu`	Copies the given GPU array to host CPU.
`chainer.backends.cuda.to_gpu`	Copies the given CPU array to the specified device.

chainer.backends.cuda.copy

`chainer.backends.cuda.copy`(*array*, *out=None*, *out_device=None*, *stream=None*)

Copies a `cupy.ndarray` object using the default stream.

This function can copy the device array to the destination array on another device.

Parameters

- **array** (`cupy.ndarray`) – Array to be copied.
- **out** (`cupy.ndarray`) – Destination array. If it is not `None`, then `out_device` argument is ignored.
- **out_device** – Destination device specifier. Actual device object is obtained by passing this value to `get_device()`.
- **stream** (`cupy.cuda.Stream`) – CUDA stream.

Returns

Copied array.

If `out` is not specified, then the array is allocated on the device specified by `out_device` argument.

Return type cupy.ndarray

chainer.backends.cuda.to_cpu

`chainer.backends.cuda.to_cpu`(*array*, *stream=None*)

Copies the given GPU array to host CPU.

Parameters

- **array** (*array*, None, list or tuple) – Array or arrays to be sent to CPU.
- **stream** (`cupy.cuda.Stream`) – CUDA stream.

Returns

Array on CPU.

If some of the arrays are already on CPU, then this function just returns those arrays without performing any copy.

If input arrays include *None*, it is returned as *None* as is.

Return type numpy.ndarray, list or tuple

chainer.backends.cuda.to_gpu

`chainer.backends.cuda.to_gpu`(*array*, *device=None*, *stream=None*)

Copies the given CPU array to the specified device.

Parameters

- **array** (*array*, None, list or tuple) – Array or arrays to be sent to GPU.
- **device** – CUDA device specifier. If None or cuda.DummyDevice, the arrays will be copied to the current CUDA device.
- **stream** (*Stream*) – *(deprecated since v3.0.0)* CUDA stream. If not None, the copy runs asynchronously.

Returns

Array or arrays on GPU.

If some of the arrays are already on GPU, then this function just returns those arrays without performing any copy.

If input arrays include *None*, it is returned as *None* as is.

Return type cupy.ndarray, list or tuple

Kernel definition utilities

chainer.backends.cuda.memoize	Makes a function memoizing the result for each argument and device.
chainer.backends.cuda.clear_memo	Clears the memoized results for all functions decorated by memoize.
chainer.backends.cuda.elementwise	Creates an elementwise kernel function.
chainer.backends.cuda.raw	Creates a raw kernel function.
chainer.backends.cuda.reduce	Creates a global reduction kernel function.

chainer.backends.cuda.memoize

chainer.backends.cuda.**memoize**(*for_each_device=False*)

Makes a function memoizing the result for each argument and device.

This is a similar version of cupy.memoize(). The difference is that this function can be used in the global scope even if CUDA is not available. In such case, this function does nothing.

Note: This decorator acts as a dummy if CUDA is not available. It cannot be used for general purpose memoization even if for_each_device is set to False.

chainer.backends.cuda.clear_memo

chainer.backends.cuda.**clear_memo**()

Clears the memoized results for all functions decorated by memoize.

This function works like cupy.clear_memo() as a counterpart for *chainer.backends.cuda.memoize()*. It can be used even if CUDA is not available. In such a case, this function does nothing.

chainer.backends.cuda.elementwise

chainer.backends.cuda.**elementwise**(*in_params*, *out_params*, *operation*, *name*, ***kwargs*)

Creates an elementwise kernel function.

This function uses `memoize()` to cache the kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.

The arguments are the same as those for `cupy.ElementwiseKernel`, except that the `name` argument is mandatory.

chainer.backends.cuda.raw

chainer.backends.cuda.**raw**(*code*, *name*, **args*, ***kwargs*)

Creates a raw kernel function.

This function uses `memoize()` to cache the resulting kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.

The arguments are the same as those for `cupy.RawKernel`.

chainer.backends.cuda.reduce

chainer.backends.cuda.**reduce**(*in_params*, *out_params*, *map_expr*, *reduce_expr*, *post_map_expr*, *identity*, *name*, ***kwargs*)

Creates a global reduction kernel function.

This function uses `memoize()` to cache the resulting kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.

The arguments are the same as those for `cupy.ReductionKernel`, except that the `name` argument is mandatory.

CPU/GPU generic code support

`chainer.backends.cuda.get_array_module`	Gets an appropriate one from `numpy` or `cupy`.

chainer.backends.cuda.get_array_module

chainer.backends.cuda.**get_array_module**(**args*)

Gets an appropriate one from `numpy` or `cupy`.

This is almost equivalent to `cupy.get_array_module()`. The differences are that this function can be used even if CUDA is not available and that it will return their data arrays' array module for `Variable` arguments.

Deprecated since version v5.0.0: This API is deprecated. Please use `get_array_module()` instead.

Parameters **args** – Values to determine whether NumPy or CuPy should be used.

Returns `cupy` or `numpy` is returned based on the types of the arguments.

Return type module

cuDNN support

chainer.backends.cuda.set_max_workspace_size	Sets the workspace size for cuDNN.
chainer.backends.cuda.get_max_workspace_size	Gets the workspace size for cuDNN.

chainer.backends.cuda.set_max_workspace_size

chainer.backends.cuda.**set_max_workspace_size**(*size*)

Sets the workspace size for cuDNN.

Check "cuDNN Library User Guide" for detail.

Parameters **size** – The workspace size for cuDNN.

chainer.backends.cuda.get_max_workspace_size

chainer.backends.cuda.**get_max_workspace_size**()

Gets the workspace size for cuDNN.

Check "cuDNN Library User Guide" for detail.

Returns The workspace size for cuDNN.

Return type int

4.12.4 Intel64 (iDeep)

iDeep is a module that provides NumPy-like API and DNN acceleration using MKL-DNN for Intel CPUs. See *Tips and FAQs* and *Performance Best Practices* for details.

chainer.backends.intel64.is_ideep_available	Returns if iDeep is available.

chainer.backends.intel64.is_ideep_available

chainer.backends.intel64.**is_ideep_available**()

Returns if iDeep is available.

Returns `True` if the supported version of iDeep is installed.

Return type bool

4.12.5 ChainerX

chainer.backend.from_chx	Converts an array or arrays from ChainerX to NumPy or CuPy ones.
chainer.backend.to_chx	Converts an array or arrays to ChainerX.

chainer.backend.from_chx

`chainer.backend.from_chx(array)`

Converts an array or arrays from ChainerX to NumPy or CuPy ones.

Destination array types are chosen such that no copies occur.

chainer.backend.to_chx

`chainer.backend.to_chx(array)`

Converts an array or arrays to ChainerX.

Destination ChainerX devices are chosen according to the types of input arrays.

4.13 Utilities

4.13.1 Convolution/Deconvolution utilities

chainer.utils.get_conv_outsize	Calculates output size of convolution.
chainer.utils.get_deconv_outsize	Calculates output size of deconvolution.

chainer.utils.get_conv_outsize

`chainer.utils.get_conv_outsize(size, k, s, p, cover_all=False, d=1)`

Calculates output size of convolution.

This function takes the size of input feature map, kernel, stride, and pooling of one particular dimension, then calculates the output feature map size of that dimension.

See also:

get_deconv_outsize()

Parameters

- **size** (*int*) – The size of input feature map. It usually is the length of a side of feature map.
- **k** (*int*) – The size of convolution kernel.
- **s** (*int*) – The size of stride.
- **p** (*int*) – The size of padding.
- **cover_all** (*bool*) – Use `cover_all` option or not.
- **d** (*int*) – The size of dilation.

Returns The expected output size of the convolution operation.

Return type int

chainer.utils.get_deconv_outsize

`chainer.utils.get_deconv_outsize(size, k, s, p, cover_all=False, d=1)`

Calculates output size of deconvolution.

This function takes the size of input feature map, kernel, stride, and pooling of one particular dimension, then calculates the output feature map size of that dimension.

See also:

get_conv_outsize()

Parameters

- **size** (*int*) – The size of input feature map. It usually is the length of a side of feature map.
- **k** (*int*) – The size of deconvolution kernel.
- **s** (*int*) – The size of stride.
- **p** (*int*) – The size of padding.
- **cover_all** (*bool*) – Use cover_all option or not.
- **d** (*int*) – The size of dilation.

Returns The expected output size of the deconvolution operation.

Return type int

4.13.2 Common algorithms

chainer.utils.WalkerAlias	Implementation of Walker's alias method.

chainer.utils.WalkerAlias

class chainer.utils.**WalkerAlias**(*probs*)

Implementation of Walker's alias method.

This method generates a random sample from given probabilities $p_1, \dots, p_n$ in $O(1)$ time. It is more efficient than choice(). This class works on both CPU and GPU.

Parameters probs (*float list*) – Probabilities of entries. They are normalized with *sum(probs)*.

See: Wikipedia article

Methods

device_resident_accept(*visitor*)

Applies the visitor to all the device objects in this instance.

Parameters visitor (DeviceResidentsVisitor) – Visitor.

This method should be overridden if the concrete class has custom sub-hierarchy of device resident objects.

from_chx()

Converts parameter variables and persistent values from ChainerX to NumPy/CuPy devices without any copy.

sample(*shape*)

Generates a random sample based on given probabilities.

Parameters shape (*tuple of int*) – Shape of a return value.

Returns Returns a generated array with the given shape. If a sampler is in CPU mode the return value is a `numpy.ndarray` object, and if it is in GPU mode the return value is a `cupy.ndarray` object.

sample_gpu(*shape*)

sample_xp(*xp*, *shape*)

to_chx()

Converts parameter variables and persistent values to ChainerX without any copy.

This method does not handle non-registered attributes. If some of such attributes must be copied to ChainerX, the link implementation must override this method to do so.

Returns: self

to_cpu()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation should override `device_resident_accept()` to do so.

Returns: self

to_device(*device*)

Copies parameter variables and persistent values to the specified device.

This method does not handle non-registered attributes. If some of such attributes must be copied to the device, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. See `get_device()` for available values.

Returns: self

to_gpu(*device=None*)

Copies parameter variables and persistent values to GPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override `device_resident_accept()` to do so.

> **Warning:** This method does not transfer the parameters if they are already on GPU. Use `to_device` to perform inter-GPU transfer.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self

to_intel64()

Copies parameter variables and persistent values to CPU.

Deprecated since version v7.0.0: Use `to_device()` instead.

__eq__()

Return self==value.

__ne__()

Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

device
`Device` instance.

use_gpu

xp
Array module corresponding to the device.

Depending on the device in which this object resides, this property returns `numpy`, `cupy` or `chainerx`.

4.13.3 Common utilities

`chainer.print_runtime_info`	Shows Chainer runtime information.

chainer.print_runtime_info

chainer.**print_runtime_info**(*out=None*)
Shows Chainer runtime information.

Runtime information includes:

- OS platform
- Chainer version
- ChainerX version
- NumPy version
- CuPy version
  - CUDA information
  - cuDNN information
  - NCCL information
- iDeep version

Parameters **out** – Output destination. If it is `None`, runtime information will be shown in `sys.stdout`.

4.13.4 Reporter

chainer.Reporter	Object to which observed values are reported.
chainer.get_current_reporter	Returns the current reporter object.
chainer.report	Reports observed values with the current reporter object.
chainer.report_scope	Returns a report scope with the current reporter.

chainer.Reporter

class chainer.**Reporter**

Object to which observed values are reported.

Reporter is used to collect values that users want to watch. The reporter object holds a mapping from value names to the actually observed values. We call this mapping *observations*.

When a value is passed to the reporter, an object called *observer* can be optionally attached. In this case, the name of the observer is added as the prefix of the value name. The observer name should be registered beforehand.

See the following example:

```
>>> from chainer import Reporter, report, report_scope
>>>
>>> reporter = Reporter()
>>> observer = object()  # it can be an arbitrary (reference) object
>>> reporter.add_observer('my_observer', observer)
>>> observation = {}
>>> with reporter.scope(observation):
...     reporter.report({'x': 1}, observer)
...
>>> observation
{'my_observer/x': 1}
```

There are also a global API to add values:

```
>>> observation = {}
>>> with report_scope(observation):
...     report({'x': 1}, observer)
...
>>> observation
{'my_observer/x': 1}
```

The most important application of Reporter is to report observed values from each link or chain in the training and validation procedures. *Trainer* and some extensions prepare their own Reporter object with the hierarchy of the target link registered as observers. We can use *report()* function inside any links and chains to report the observed values (e.g., training loss, accuracy, activation statistics, etc.).

Variables observation – Dictionary of observed values.

Methods

__enter__()

Makes this reporter object current.

__exit__(*exc_type*, *exc_value*, *traceback*)

Recovers the previous reporter object to the current.

add_observer(*name*, *observer*)
Registers an observer of values.

Observer defines a scope of names for observed values. Values observed with the observer are registered with names prefixed by the observer name.

Parameters

- **name** (*str*) – Name of the observer.
- **observer** – The observer object. Note that the reporter distinguishes the observers by their object ids (i.e., `id(owner)`), rather than the object equality.

add_observers(*prefix*, *observers*)
Registers multiple observers at once.

This is a convenient method to register multiple objects at once.

Parameters

- **prefix** (*str*) – Prefix of each name of observers.
- **observers** – Iterator of name and observer pairs.

report(*values*, *observer=None*)
Reports observed values.

The values are written with the key, prefixed by the name of the observer object if given.

Note: If a value is of type *Variable*, the variable is copied without preserving the computational graph and the new variable object purged from the graph is stored to the observer. This behavior can be changed by setting `chainer.config.keep_graph_on_report` to `True`.

Parameters

- **values** (*dict*) – Dictionary of observed values.
- **observer** – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

scope(*observation*)
Creates a scope to report observed values to `observation`.

This is a context manager to be passed to `with` statements. In this scope, the observation dictionary is changed to the given one.

It also makes this reporter object current.

Parameters **observation** (*dict*) – Observation dictionary. All observations reported inside of the `with` statement are written to this dictionary.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.get_current_reporter

chainer.**get_current_reporter**()
Returns the current reporter object.

chainer.report

chainer.**report**(*values*, *observer=None*)
Reports observed values with the current reporter object.

Any reporter object can be set current by the `with` statement. This function calls the `Reporter.report()` method of the current reporter. If no reporter object is current, this function does nothing.

Example

The most typical example is a use within links and chains. Suppose that a link is registered to the current reporter as an observer (for example, the target link of the optimizer is automatically registered to the reporter of the `Trainer`). We can report some values from the link as follows:

```
class MyRegressor(chainer.Chain):
    def __init__(self, predictor):
        super(MyRegressor, self).__init__(predictor=predictor)

    def __call__(self, x, y):
        # This chain just computes the mean absolute and squared
        # errors between the prediction and y.
        pred = self.predictor(x)
        abs_error = F.sum(abs(pred - y)) / len(x)
        loss = F.mean_squared_error(pred, y)

        # Report the mean absolute and squared errors.
        chainer.report({
            'abs_error': abs_error,
            'squared_error': loss,
        }, self)

        return loss
```

If the link is named `'main'` in the hierarchy (which is the default name of the target link in the `StandardUpdater`), these reported values are named `'main/abs_error'` and `'main/squared_error'`. If these values are reported inside the `Evaluator` extension, `'validation/'` is added at the head of the link name, thus the item names are changed to `'validation/main/abs_error'` and `'validation/main/squared_error'` (`'validation'` is the default name of the Evaluator extension).

Parameters

- **values** (`dict`) – Dictionary of observed values.

- **observer** – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

chainer.report_scope

chainer.**report_scope**(*observation*)
Returns a report scope with the current reporter.

This is equivalent to `get_current_reporter().scope(observation)`, except that it does not make the reporter current redundantly.

4.13.5 Summary and DictSummary

chainer.Summary	Online summarization of a sequence of scalars.
chainer.DictSummary	Online summarization of a sequence of dictionaries.

chainer.Summary

class chainer.**Summary**
Online summarization of a sequence of scalars.

Summary computes the statistics of given scalars online.

Methods

add(*value*, *weight=1*)
Adds a scalar value.

Parameters

- **value** – Scalar value to accumulate. It is either a NumPy scalar or a zero-dimensional array (on CPU or GPU).
- **weight** – An optional weight for the value. It is a NumPy scalar or a zero-dimensional array (on CPU or GPU). Default is 1 (integer).

compute_mean()
Computes the mean.

make_statistics()
Computes and returns the mean and standard deviation values.

Returns Mean and standard deviation values.

Return type tuple

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.DictSummary

class chainer.**DictSummary**
Online summarization of a sequence of dictionaries.

`DictSummary` computes the statistics of a given set of scalars online. It only computes the statistics for scalar values and variables of scalar values in the dictionaries.

Methods

add(*d*)
Adds a dictionary of scalars.

Parameters d (*dict*) – Dictionary of scalars to accumulate. Only elements of scalars, zero-dimensional arrays, and variables of zero-dimensional arrays are accumulated. When the value is a tuple, the second element is interpreted as a weight.

compute_mean()
Creates a dictionary of mean values.

It returns a single dictionary that holds a mean value for each entry added to the summary.

Returns Dictionary of mean values.

Return type dict

make_statistics()
Creates a dictionary of statistics.

It returns a single dictionary that holds mean and standard deviation values for every entry added to the summary. For an entry of name `'key'`, these values are added to the dictionary by names `'key'` and `'key.std'`, respectively.

Returns Dictionary of statistics of all entries.

Return type dict

serialize(*serializer*)

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.13.6 Sparse utilities

A *chainer.Variable* can be converted into a sparse matrix in e.g. COO (Coordinate list) format. A sparse matrix stores the same data as the original object but with a different internal representation, optimized for efficient operations on sparse data, i.e. data with many zero elements.

Following are a list of supported sparse matrix formats and utilities for converting between a *chainer.Variable* and these representations.

Note: Please be aware that only certain functions accept sparse matrices as inputs, such as *chainer.functions.sparse_matmul()*.

chainer.utils.CooMatrix	A sparse matrix in COO format.
chainer.utils.to_coo	Returns a single or a batch of matrices in COO format.

chainer.utils.CooMatrix

class chainer.utils.**CooMatrix**(*data*, *row*, *col*, *shape*, *order=None*, *requires_grad=False*)
A sparse matrix in COO format.

Parameters

- **data** (*N-dimensional array*) – The entries of the matrix. The entries are usually non-zero-elements in the matrix.
- **row** (*N-dimensional array*) – The row indices of the matrix entries.
- **col** (*N-dimensional array*) – The column indices of the matrix entries.
- **shape** (*tuple of int*) – The shape of the matrix in dense format.
- **order** (*'C', 'F', 'other' or None*) – If 'C', the maxtix is assumed that its row indices are sorted. If 'F', the matrix is assumed that its column indices are sorted. If 'other', the matrix is assumed as neither 'C' order nor 'F' order. If None (this is the default), the matrix is automatically checked if it is 'C' order, 'F' order or another. This information will be used by some functions like *sparse_matmul()* as a hint to improve performance.
- **requires_grad** (*bool*) – If True, gradient of this sparse matrix will be computed in back-propagation.

See also:

See *to_coo()* for how to construct a COO matrix from an array.

Methods

to_dense()
Returns a dense matrix format of this sparse matrix.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

chainer.utils.to_coo

`chainer.utils.`**`to_coo`**(*x*, *ldnz=None*, *requires_grad=False*)
Returns a single or a batch of matrices in COO format.

Parameters

- **x** (*N-dimensional array*) – Input dense matrix. The ndim of `x` must be two or three. If ndim is two, it is treated as a single matrix. If three, it is treated as batched matrices.
- **ldnz** (*int*) – Size of arrays for data, row index and column index to be created. The Actual size becomes max(nnz, ldnz) where nnz is number of non-zero elements in a input dense matrix.
- **requires_grad** (*bool*) – If `True`, gradient of sparse matrix will be computed in back-propagation.

Returns A sparse matrix or batched sparse matrices in COO format of a given dense matrix or batched dense matrices.

Return type *CooMatrix*

Example

Create a `CooMatrix` from an array with 2 non-zero elements and 4 zeros and access its attributes. No batch dimension is involved.

```
>>> data = np.array([[0, 2, 0], [-1, 0, 0]], np.float32)
>>> x = chainer.utils.to_coo(data)
>>> x.data
variable([ 2., -1.])
>>> x.row
array([0, 1], dtype=int32)
>>> x.col
array([1, 0], dtype=int32)
>>> x.shape
(2, 3)
```

4.13.7 Experimental feature annotation

chainer.utils.experimental	Declares that user is using an experimental feature.

chainer.utils.experimental

chainer.utils.**experimental**(*api_name*)

Declares that user is using an experimental feature.

The developer of an API can mark it as *experimental* by calling this function. When users call experimental APIs, FutureWarning is issued. The presentation of FutureWarning is disabled by setting chainer.disable_experimental_feature_warning to True, which is False by default.

The basic usage is to call it in the function or method we want to mark as experimental along with the API name.

```
from chainer import utils

def f(x):
    utils.experimental('chainer.foo.bar.f')
    # concrete implementation of f follows

f(1)
```

```
... FutureWarning: chainer.foo.bar.f is experimental. The interface can change in␣
↪the future. ...
```

We can also make a whole class experimental. In that case, we should call this function in its `__init__` method.

```
class C():
    def __init__(self):
      utils.experimental('chainer.foo.C')

C()
```

```
... FutureWarning: chainer.foo.C is experimental. The interface can change in the␣
↪future. ...
```

If we want to mark `__init__` method only, rather than class itself, it is recommended that we explicitly feed its API name.

```
class D():
    def __init__(self):
        utils.experimental('D.__init__')

D()
```

```
...  FutureWarning: D.__init__ is experimental. The interface can change in the␣
↪future. ...
```

Currently, we do not have any sophisticated way to mark some usage of non-experimental function as experimental. But we can support such usage by explicitly branching it.

```
def g(x, experimental_arg=None):
    if experimental_arg is not None:
        utils.experimental('experimental_arg of chainer.foo.g')
```

Parameters **api_name** (*str*) – The name of an API marked as experimental.

4.14 Configuring Chainer

Chainer provides some global settings that affect the behavior of some functionalities. Such settings can be configured using the *unified configuration system*. The system provides a transparent way to manage the configuration for each process and for each thread.

The configuration is managed by two global objects: `chainer.global_config` and `chainer.config`.

- The `global_config` object maintains the configuration shared in the Python process. This is an instance of the `GlobalConfig` class. It can be used just as a plain object, and users can freely set any attributes on it.
- The `config` object, on the other hand, maintains the configuration for the current thread. This is an instance of the `LocalConfig` class. It behaves like a thread-local object, and any attribute modifications are only visible to the current thread.

If no value is set to `config` for a given key, `global_config` is transparently referred. Thanks to this transparent lookup, users can always use `config` to read any configuration so that the thread-local configuration is used if available and otherwise the default global setting is used.

The following entries of the configuration are currently provided by Chainer. Some entries support environment variables to set the default values. Note that the default values are set in the global config.

4.14.1 Configuration Keys

- **`cudnn_deterministic` (default: `False`)** Flag to configure deterministic computations in cuDNN APIs.

  If it is `True`, convolution functions that use cuDNN use the deterministic mode (i.e, the computation is reproducible). Otherwise, the results of convolution functions using cuDNN may be non-deterministic in exchange for better performance.

- **`debug` (default: `False`)** Debug mode flag.

  If it is `True`, Chainer runs in debug mode. Enabling debug mode may introduce some performance overhead. See *Debug Mode* for more information of the debug mode.

  You can change the default value to `True` by setting `CHAINER_DEBUG` environment variable to `1`.

- **`dtype` (default: `numpy.float32`)** Default floating point data type.

  Chainer uses this dtype to construct arrays when the dtype is not specified (e.g. initializers).

  You can change the default value by setting `CHAINER_DTYPE` environment variable to `mixed16`, `float16`, `float32`, `float64`.

  Note: If you want to use float16 for better performance, it is recommended that you use `mixed16` instead of `float16`.

- **`enable_backprop` (default: `True`)** Flag to enable backpropagation support.

  If it is `True`, computational graphs are created during forward passes by `FunctionNode`s, allowing backpropagation to start from any `Variable` in the graph. Otherwise, computational graphs are not created but memory consumptions are reduced. So calling `backward()` on the results of a function will not compute any gradients of any input.

- **`keep_graph_on_report` (default: `False`)** Flag to configure whether or not to let `report()` keep the computational graph.

  If it is `False`, `report()` does not keep the computational graph when a `Variable` object is reported. It means that `report()` stores a copy of the `Variable` object which is purged from the computational graph. If it is `True`, `report()` just stores the `Variable` object as is with the computational graph left attached.

You can change the default value to `True` by setting `CHAINER_KEEP_GRAPH_ON_REPORT` environment variable to `1`.

- **`warn_nondeterministic` (default: `False`)** Flag to give warning when a non-deterministic function is used. This function is experimental.

  If it is true, then functions that use non-deterministic functions and cannot be given a seed, such as atomicAdd, will give a warning when executed. For functions that can take a seed argument, such as `split_dataset_random()`, setting the seed should be done when the function is called and will not be flagged by this setting.

  Note that this feature is provided as best-effort. It cannot assure that every nondeterministic function can be detected. For example, SSE computations in CPU mode may cause non-deterministic behavior that would not raise a warning.

  Also, determinisitic outputs may still result, even if this flag produces a non-deterministic warning. For example, reduction on 1-dim axis should always be deterministic, but it may raise a warning.

- **`train` (default: `True`)** Training mode flag.

  If it is `True`, Chainer runs in training mode. Otherwise, it runs in the testing (evaluation) mode.

  This configuration is used by Functions and Links that need to behave differently between training phase and evaluation (inference) phase. One example is `chainer.links.BatchNormalization` updates statistics using input data only when `train` is set to `True`. The other example is `chainer.functions.dropout()`, which does nothing when `train` is set to `False`.

  Generally, you are responsible to change the configuration to `False` during evaluation. If you are using `Trainer` with `Evaluator` extension, `train` configuration will automatically be switched to `False` during evaluation in the training loop.

  Note that this parameter does not reduce memory consumption or affect the creation of computational graphs required in order to compute gradients.

- **`type_check` (default: `True`)** Type checking mode flag.

  If it is `True`, Chainer checks the types (data types and shapes) of inputs on `Function` applications. Otherwise, it skips type checking.

  You can change the default value to `False` by setting `CHAINER_TYPE_CHECK` environment variable to `0`.

- **`use_cudnn` (default: `'auto'`)** Flag to configure whether or not to use cuDNN.

  This is a ternary flag with `'always'`, `'auto'`, and `'never'` as its allowed values. The meaning of each flag is as follows.

  - If it is `'always'`, Chainer will try to use cuDNN everywhere if possible.
  - If it is `'auto'`, Chainer will use cuDNN only if it is known that the usage does not degrade the performance.
  - If it is `'never'`, Chainer will never use cuDNN anywhere.

  You can change the default value by setting `CHAINER_USE_CUDNN` environment variable to any of `'always'`, `'auto'` or `'never'`.

- **`use_ideep` (default: `'never'`)** Flag to configure whether or not to use iDeep.

  This is a ternary flag with `'always'`, `'auto'`, and `'never'` as its allowed values. The meaning of each flag is as follows.

  - If it is `'always'`, Chainer will try to use iDeep everywhere if possible.
  - If it is `'auto'`, Chainer will use iDeep only if it is known that the usage does not degrade the performance.
  - If it is `'never'`, Chainer will never use iDeep anywhere.

  You can change the default value by setting `CHAINER_USE_IDEEP` environment variable to any of `'always'`, `'auto'` or `'never'`.

  Note that in spite of the configuration, optimizers will use iDeep if and only if the link is converted manually to iDeep (e.g., `model.to_intel64()`).

- **`lazy_grad_sum` (default: `False`)** Flag to control the behavior of gradient accumulation.

  If it is `True`, gradients are accumulated in batch for performance. Otherwise gradients are accumulated one by one.

  You can change the default value to `True` by setting `CHAINER_LAZY_GRAD_SUM` environment variable to `1`.

- **`use_cudnn_tensor_core` (default: `'auto'`)** Flag to configure whether or not to enable Tensor Core operatons in cuDNN.

  This is a ternary flag with `'always'`, `'auto'`, and `'never'` as its allowed values. The meaning of each flag is as follows.

  - If it is `always`, Chainer uses cuDNN's Tensor Core operations.
  - If it is `never`, Chainer does not use cuDNN's Tensor Core operations.
  - If it is `auto`, Chainer checks cuDNN version, the data type of input, the compute capability of the GPU used, and configures whether or not to use cuDNN's Tensor Core operations.

- **`autotune` (default: `False`)** Autotune for convolutional networks flag.

  If it is `True`, Chainer uses the cuDNN autotune feature to find the fastest calculation process for *`chainer.links.Convolution2D`*, *`ConvolutionND`*, *`Deconvolution2D`*, or *`DeconvolutionND`* links.

- **`cudnn_fast_batch_normalization` (default: `False`)** Flag to configure whether or not to enable use of fast implementation for batch normalization in cuDNN.

  If `True`, Chainer will try to use the fast implementation for batch normalization in cuDNN by setting cuDNN's batch normalization mode to `CUDNN_BATCHNORM_SPATIAL_PERSISTENT`. You can change the default value to `True` by setting `CHAINER_CUDNN_FAST_BATCH_NORMALIZATION` environment variable to `1`.

- **`in_recomputing` (default: `False`)** This flag is automatically set by *`chainer.functions.forget()`* and not intended to be changed by users. You can use this flag when implementing your own Link to avoid updating the internal states during recomputation done by *`chainer.functions.forget()`*. See the documentation of *`chainer.functions.forget()`* for details.

- **`use_static_graph` (default: `True`)** Flag to configure whether or not to use the static subgraph optimization feature. Where the static subgraph optimization decorator is used, we generally assume that the feature should be used and the default value is thus `True`. However, if you would want to run the same code without the feature, you can simply set the flag to `False` instead of removing the decorators. This is useful when for instance running your model with ChainerX, since ChainerX is not supported by the static subgraph optimization feature.

4.14.2 User-defined Keys

Users can also define their own configurations. There are two ways:

1. Use Chainer's configuration objects. In this case, **it is strongly recommended that the name be prefixed by "user_"** to avoid name conflicts with configurations introduced to Chainer in the future.
2. Use your own configuration objects. Users can define their own configuration objects using *chainer.configuration.GlobalConfig* and *chainer.configuration.LocalConfig*. In this case, there is no need to take care of the name conflicts.

4.14.3 Changing Configuration

If you want to share a setting within the process, set an attribute to the global configuration. This value is automatically extracted by referring to the local config.

```
>>> chainer.global_config.train
True
>>> chainer.config.train
True

>>> chainer.global_config.train = False

>>> chainer.global_config.train
False
>>> chainer.config.train
False
```

If you set an attribute to the local configuration, the value is only visible to the current thread.

```
>>> chainer.global_config.train
True
>>> chainer.config.train
True

>>> chainer.config.train = False

>>> chainer.global_config.train
True
>>> chainer.config.train
False
```

If you want to temporarily modify the configuration for the specific scope, you can use *using_config()*. For example, if you only want to enable debug mode in a fragment of code, write as follows.

```
>>> with chainer.using_config('debug', True):
...     pass  # code running in debug mode
```

If you want to switch to the test mode for an evaluation, you can do that in the same way.

```
>>> # Do training here
>>> with chainer.using_config('train', False):
...     pass  # Perform evaluation here
```

Note that *Evaluator* automatically switches to the test mode, and thus you do not need to manually switch in the loss function for the evaluation.

You can also make your own code behave differently in training and test modes as follows.

```
if chainer.config.train:
    pass  # code only running in the training mode
else:
    pass  # code only running in the test mode
```

chainer.global_config	
chainer.config	Thread-local configuration of Chainer.
chainer.using_config	Context manager to temporarily change the thread-local configuration.
chainer.configuration.GlobalConfig	
chainer.configuration.LocalConfig	Thread-local configuration of Chainer.

chainer.global_config

chainer.**global_config = <chainer.configuration.GlobalConfig object>**

chainer.config

chainer.**config = <chainer.configuration.LocalConfig object>**

Thread-local configuration of Chainer.

This class implements the local configuration. When a value is set to this object, the configuration is only updated in the current thread. When a user tries to access an attribute and there is no local value, it automatically retrieves a value from the global configuration.

chainer.using_config

chainer.**using_config**(*name*, *value*, *config=chainer.config*)

Context manager to temporarily change the thread-local configuration.

Parameters

- **name** (*str*) – Name of the configuration to change.
- **value** – Temporary value of the configuration entry.
- **config** (LocalConfig) – Configuration object. Chainer's thread-local configuration is used by default.

See also:

Configuring Chainer

chainer.configuration.GlobalConfig

class chainer.configuration.**GlobalConfig**

Methods

show(*file=sys.stdout*)

Prints the global config entries.

The entries are sorted in the lexicographical order of the entry name.

Parameters **file** – Output file-like object.

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

autotune = None

cudnn_deterministic = None

cudnn_fast_batch_normalization = None

debug = None

dtype = None

enable_backprop = None

in_recomputing = None

keep_graph_on_report = None

lazy_grad_sum = None

schedule_func = None

train = None

type_check = None

use_cudnn = None

use_cudnn_tensor_core = None

use_ideep = None

use_static_graph = None

warn_nondeterministic = None

chainer.configuration.LocalConfig

class chainer.configuration.**LocalConfig**(*global_config*)
Thread-local configuration of Chainer.

This class implements the local configuration. When a value is set to this object, the configuration is only updated in the current thread. When a user tries to access an attribute and there is no local value, it automatically retrieves a value from the global configuration.

Methods

show (*file=sys.stdout*)
Prints the config entries.

The entries are sorted in the lexicographical order of the entry names.

Parameters **file** – Output file-like object.

Example

You can easily print the list of configurations used in the current thread.

```
>>> chainer.config.show()
debug           False
enable_backprop True
train           True
type_check      True
```

__eq__ ()
Return self==value.

__ne__ ()
Return self!=value.

__lt__ ()
Return self<value.

__le__ ()
Return self<=value.

__gt__ ()
Return self>value.

__ge__ ()
Return self>=value.

Related functions

chainer.get_dtype	Resolves Chainer's default dtype.
chainer.mixed16	Dtype-like object that represents 16/32 bits mixed precision float.

chainer.get_dtype

chainer.**get_dtype** (*dtype=None*, *map_mixed16=None*)
Resolves Chainer's default dtype.

Parameters

- **dtype** – Dtype specifier. If this value is specified (not None), this function returns the dtype object corresponding to it.
- **map_mixed16** – Dtype specifier. When chainer.config.dtype is mixed16, this option is used. If this value is None, float16 is used.

Returns If dtype is not None, it returns the dtype normalized by numpy.dtype(). Otherwise, it returns chainer.config.dtype (see *Configuring Chainer*) normalized as well. When chainer.config.dtype is *mixed16* and map_mixed16 is specified, it returns the normalized version of map_mixed16.

chainer.mixed16

chainer.**mixed16 = dtype('mixed16')**
Dtype-like object that represents 16/32 bits mixed precision float.

4.14.4 Environment Variables

Here are the environment variables Chainer uses.

CHAINER_SEED	Default seed value of random number generators for CUDA. If it is not set, the seed value is generated from Python random module. Set an integer value in decimal format.
CHAINER_DATASET_ROOT	Default directory path to store the downloaded datasets. See *Datasets* for details.
CHAINER_CUDNN	Set 0 to completely disable cuDNN in Chainer. In this case, cuDNN will not be used regardless of CHAINER_USE_CUDNN and chainer.config.use_cudnn configuration. Otherwise cuDNN is enabled automatically.
CHAINER_USE_CUDNN	Used as the default value for chainer.config.use_cudnn configuration. The value must be any of 'always', 'auto' or 'never'. If CHAINER_CUDNN is set to 0, this environment variable has no effect. See *Configuring Chainer* for details.
CHAINER_CUDNN_FAST_BATCH_NORMALIZATION	Used as the default value for chainer.config.cudnn_fast_batch_normalization configuration. Set 1 to enable use of fast implementation for batch normalization in cuDNN. See *Configuring Chainer* for details.
CHAINER_USE_IDEEP	Used as the default value for chainer.config.use_ideep configuration. The value must be any of 'always', 'auto' or 'never'. See *Configuring Chainer* for details.
CHAINER_LAZY_GRAD_SUM	Used as the default value for chainer.config.lazy_grad_sum configuration. Set 1 to enable batch accumulation of gradients. See *Configuring Chainer* for details.
CHAINER_DTYPE	Used as the default value for chainer.config.dtype configuration. The value must be any of 'mixed16', 'float16', 'float32' or 'float64'. See *Configuring Chainer* for details.
CHAINER_TYPE_CHECK	Used as the default value for chainer.config.type_check configuration. Set 0 to disable type checking. Otherwise type checking is enabled automatically. See *Configuring Chainer* and *Type checking utilities* for details.
CHAINER_DEBUG	Used as the default value for chainer.config.debug configuration. Set 1 to enable debug mode. It is disabled by default. In debug mode, Chainer performs various runtime checks that can help debug user's code at the cost of some overhead. See *Configuring Chainer* and *Debug Mode* for details.
CHAINER_KEEP_GRAPH_ON_REPORT	Used as the default value for chainer.config.keep_graph_on_report configuration. Set 1 to let *report()* keep the computational graph. See *Configuring Chainer* for details.
CHAINER_PYTHON_350_FORCE	Set 1 to force using Chainer with Python 3.5.0. Note that Chainer does not work with Python 3.5.0. Use Python 3.5.1+ or other supported versions (see *Installation*).

The following environment variables are only effective when running unit tests.

CHAINER_TEST_GPU_LIMIT	Number of GPUs available for unit tests. When running unit test, test cases that require more GPUs than the specified value will be skipped. Set 0 to skip all test cases that require GPU. See *Unit Testing* for details.
CHAINER_TEST_RANDOM_NONDETERMINISTIC	Set 1 to use non-fixed seed for random number generators, even for test cases annotated with *fix_random*.

4.15 Debug Mode

In debug mode, Chainer checks values of variables on runtime and shows more detailed error messages. It helps you to debug your programs. However, it requires some additional overhead time.

If you want to enable debug mode for the entire code, you can set `CHAINER_DEBUG` environment variable to `1`.

You can also enable or disable debug mode for the specific scope of code with `chainer.using_config()` or by changing `chainer.config.debug` configuration.

```
with chainer.using_config('debug', True):
    ...
```

See *Configuring Chainer* for the details of Chainer's configuration mechanism.

In debug mode, Chainer checks all results of forward and backward computation, and if it finds a NaN value, it raises a `RuntimeError`. Some functions and links also check validity of input values more strictly.

You can check if debug mode is enabled with `chainer.is_debug()` function.

`chainer.is_debug`	Returns if the debug mode is enabled or not in the current thread.
`chainer.set_debug`	Enables or disables the debug mode in the current thread.

4.15.1 chainer.is_debug

`chainer.is_debug()`

Returns if the debug mode is enabled or not in the current thread.

Returns `True` if the debug mode is enabled.

Return type bool

4.15.2 chainer.set_debug

`chainer.set_debug(debug)`

Enables or disables the debug mode in the current thread.

Note: `chainer.set_debug(value)` is equivalent to `chainer.config.debug = value`.

Parameters **debug** (*bool*) – New debug mode.

4.16 Visualization of Computational Graph

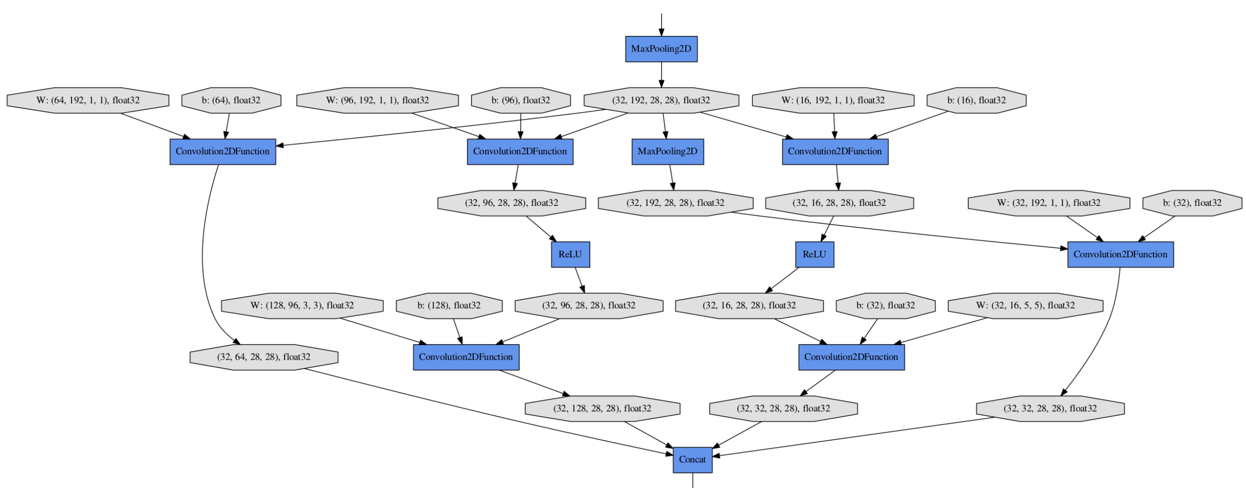
As neural networks get larger and complicated, it gets much harder to confirm if their architectures are constructed properly. Chainer supports visualization of computational graphs. Users can generate computational graphs by invoking `build_computational_graph()`. Generated computational graphs are dumped to specified format (Currently Dot Language is supported).

Basic usage is as follows:

```
import chainer.computational_graph as c
...
g = c.build_computational_graph(vs)
with open('path/to/output/file', 'w') as o:
    o.write(g.dump())
```

where `vs` is list of `Variable` instances and `g` is an instance of `ComputationalGraph`. This code generates the computational graph that are backward-reachable (i.e. reachable by repetition of steps backward) from at least one of `vs`.

Here is an example of (a part of) the generated graph (inception(3a) in GoogLeNet). This example is from `example/imagenet`.



`chainer.computational_graph.build_computational_graph`	Builds a graph of functions and variables backward-reachable from outputs.
`chainer.computational_graph.ComputationalGraph`	Class that represents computational graph.

4.16.1 chainer.computational_graph.build_computational_graph

`chainer.computational_graph.`**`build_computational_graph`**`(`*outputs, remove_split=True, variable_style='default', function_style='default', rankdir='TB', remove_variable=False, show_name=True*`)`

Builds a graph of functions and variables backward-reachable from outputs.

Parameters

- **outputs** (*Variable*, *VariableNode*, *FunctionNode*, or list) – node(s) from which the graph is constructed. Each element of outputs must be either *Variable* object, *VariableNode* object, or *FunctionNode* object.
- **remove_split** (*bool*) – It must be `True`. This argument is left for backward compatibility.
- **variable_style** (*dict or 'default'*) – Dot node style for variable. Possible keys are 'shape', 'color', 'fillcolor', 'style' etc. If the special value `'default'` is specified, the default configuration will be used.
- **function_style** (*dict or 'default'*) – Dot node style for function. Possible keys are 'shape', 'color', 'fillcolor', 'style' etc. If the special value `'default'` is specified, the default configuration will be used.
- **rankdir** (*str*) – Direction of the graph that must be TB (top to bottom), BT (bottom to top), LR (left to right) or RL (right to left).
- **remove_variable** (*bool*) – If `True`, `VariableNode`s are removed from the resulting computational graph. Only `FunctionNode`s are shown in the output.
- **show_name** (*bool*) – If `True`, the `name` attribute of each node is added to the label of the node. Default is `True`.

Returns

A graph consisting of nodes and edges that are backward-reachable from at least one of `outputs`.

If `unchain_backward` was called in some variable in the computational graph before this function, backward step is stopped at this variable.

For example, suppose that computational graph is as follows:

```
    |--> f ---> y
x --+
    |--> g ---> z
```

Let `outputs = [y, z]`. Then the full graph is emitted.

Next, let `outputs = [y]`. Note that `z` and `g` are not backward-reachable from `y`. The resulting graph would be following:

```
x ---> f ---> y
```

See `TestGraphBuilder` for details.

Return type *ComputationalGraph*

Note: The default configuration for `variable_style` is `{'shape': 'octagon', 'fillcolor': '#E0E0E0', 'style': 'filled'}` and the default configuration for `function_style` is `{'shape': 'record', 'fillcolor': '#6495ED', 'style': 'filled'}`.

Note: The default behavior of `ComputationalGraph` has been changed from v1.23.0, so that it ouputs the richest representation of a graph as default, namely, styles are set and names of functions and variables are shown. To reproduce the same result as previous versions (<= v1.22.0), please specify *variable_style=None*, *function_style=None*, and *show_name=False* explicitly.

4.16.2 chainer.computational_graph.ComputationalGraph

class chainer.computational_graph.**ComputationalGraph**(*nodes*, *edges*, *variable_style='default'*, *function_style='default'*, *rankdir='TB'*, *remove_variable=False*, *show_name=True*)

Class that represents computational graph.

Note: We assume that the computational graph is directed and acyclic.

Parameters

- **nodes** (*list*) – List of nodes. Each node is either `VariableNode` object or `FunctionNode` object.
- **edges** (*list*) – List of edges. Each edge consists of pair of nodes.
- **variable_style** (dict or *'default'*) – Dot node style for variable. If the special value `'default'` is specified, the default configuration will be used.
- **function_style** (dict or *default*) – Dot node style for function. If the special value `'default'` is specified, the default configuration will be used.
- **rankdir** (*str*) – Direction of the graph that must be TB (top to bottom), BT (bottom to top), LR (left to right) or RL (right to left).
- **remove_variable** (*bool*) – If `True`, `VariableNode`s are removed from the resulting computational graph. Only `FunctionNode`s are shown in the output.
- **show_name** (*bool*) – If `True`, the `name` attribute of each node is added to the label of the node. Default is `True`.

Note: The default configuration for `variable_style` is `{'shape': 'octagon', 'fillcolor': '#E0E0E0', 'style': 'filled'}` and the default configuration for `function_style` is `{'shape': 'record', 'fillcolor': '#6495ED', 'style': 'filled'}`.

Note: The default behavior of `ComputationalGraph` has been changed from v1.23.0, so that it ouputs the richest representation of a graph as default, namely, styles are set and names of functions and variables are shown. To reproduce the same result as previous versions (<= v1.22.0), please specify *variable_style=None*, *function_style=None*, and *show_name=False* explicitly.

Methods

dump(*format='dot'*)

Dumps graph as a text.

Parameters

- **format** (*str*) – The graph language name of the output.
- **it must be 'dot'.** (*Currently,*) –

Returns The graph in specified format.

Return type str

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

4.17 Static Subgraph Optimizations: Usage

Note: This is an experimental feature and so the API might change in the future as it is developed.

This feature intends to improve runtime performance by optimizing the execution of the static subgraphs in a model. When this feature is enabled, the first iteration runs as normal except that an execution trace is also collected. The trace is then used to generate optimized code that is will be called instead of the define-by-run code starting from the second iteration.

chainer.static_graph	Decorator to mark a Chain's `__call__()` as a static sub-graph.

4.17.1 chainer.static_graph

chainer.**static_graph**(**args*, ***kwargs*)
Decorator to mark a Chain's `__call__()` as a static sub-graph.

This decorator marks the define-by-run code inside the *__call__()* method of a Chain instance as corresponding to a static computation graph or sub-graph. Such a chain will be referred to as a 'static chain'. This allows various "static graph" optimizations to be performed, which can result in significant speedups for some models.

When this decorator is used, the chain's define-by-run code executes during the first iteration as usual. However, while the define-by-run code is executing, a trace is also performed to incrementally create a corresponding static schedule. This static schedule will only contain the subset of the computations inside the define-by-run code that actually needs to run every iteration. Specifically, this will contain the code inside any functions called that were annotated with the *@static_code* decorator, which will include all Chainer built-in functions, as well as any user-defined functions that use *@static_code*. Then, starting from the second iteration, when the static chain is called, its static schedule code will be executed instead of its define-by-run code.

However, the user must also be careful of the following: - The user is responsible for applying this decorator correctly. The framework does not check that the define-by-run code corresponds to a static graph. The graph can be different between training and evaluation mode (such as when dropout and/or batch normalization are used), but should otherwise be static. - When *chainer.config.enable_backprop* is enabled, if a backward pass is not performed each iteration, then the user code must call a method *chain.schedule_manager.end_forward()‘on the static chain each iteration. - Static graphs allow tradeoffs between computation and memory usage. For example, the ‘minimize_cache_size* argument will typically result in higher memory useage when set to *False* because all cached schedules are retained. - When this feature is enabled, only the Chainer function and/or link calls inside the chain’s *__call__()* method will be included in the static schedule by default. An other code that the user puts in *__call__()*, such as a print statement or code to increment a counter for example, will not automatically get added. We will refer to such code other than Chainer function/link calls as “side-effect” code. Since side-effect code does not get included in the static schedule by default, this means that it will only every execute once, during the first iteration. There is a way to force side-effect code to be included in the static schedule, however: the user can wrapp such code inside a function that is decorated with @*static_code* to ensure that it gets added to the static schedule. For an example of this, refer to the documentation. - This feature is experimental and advanced optimizations such as kernel fusion and various memory optimizations are not implemented yet.

Usage:

This decorator should only be applied to define-by-run code that actually corresponds to a static subgraph. Refer to the documenation for additional details and examples of correct usage. This decorator should be applied to each of the largest static subgraphs in the model; it can also be applied to a static subgraph that is not the largest subgraph, but that could result in reduced performance. It is not currently allowed to mark a chain as static if it is contained within another chain that is also marked as being static. For example, suppose a static graph *A* contains a static sub-graph *B*. Then, only the chain corresponding to *A* should be marked as static and the chain corresponding to *B* should not be marked as static.

The behavior of a static chain depends on the training mode flag, *chainer.config.train*. If it is *True*, then a static chain that is called multiple times will try to use a distinct static schedule object (that is, call a distinct instance of a FunctionNode that implements that static schedule) on each call. The same schedule instance cannot be reused until the forward pass has completed, which is signaled by performing a backward pass through the model. It is therefore important that the backward pass be performed after each forward pass during training. Since this is usually the case, most usages of static chain will not required any modifications to existing code other than applying this decorator. However, if you would like to perform multiple forward passes during training before performing a backward pass, then you must call *chain.schedule_manager.end_forward()* after the end of each forward pass.

If test mode is active (*chainer.config.train* is *False*) then it is not necessary to inform the chain at the end of each forward pass because in test mode, a static chain always attempts to reuse existing static schedule objects. The same static schedule can be reused during a single forward pass, because it is not necessary to compute gradients. It is also possible to disable static optimzations while in test mode by setting the decorator argument *force_test_define_by_run=True*.

Note: If either ‘chainer.config.enable_backprop’ or ‘chainer.config.train’ is set to ‘False’, then cached static schedules will be reused when possible to reduce memory usage.

Double-backprop: Double-backpropagation is not enabled by default. It can be enabled by supplying the keyword argument `enable_double_backprop=True` to this decorator. Note: this feature has not been tested yet.

Restrictions on input arguments and return values of a static chain: Recall that unlike a function, there is no restrictions on the arguments to a chain. However, there currently are some restrictions when a static chain is used. Specifically, the arguments to a static chain must consist of a variable, list or tuple. In the case of a list or tuple, the elements are required to be an instance of variable, list, or tuple. There can be an arbitrary number of nested lists/ tuples. No other object types are allowed. In addition, keyword arguments are not allowed. The return value of a static chain must be a variable, list, or tuple in which each element of the list or tuple is also a variable, list, or tuple.

This decorator can be supplied with the following optional keyword arguments. This is an experimental feature, and the API and arguments might change

Parameters

- **force_test_define_by_run** (*bool*) – If *True*, disable static graph optimizations during test mode (that is, when *chainer.config.train* is False). This may be needed in order for some existing RNN links such as LSTM to work correctly, since some existing links do not correspond to a static graph in some cases. The default is *False*.
- **minimize_cache_size** (*bool*) – If *True*, minimize the number of cached static schedules in order to reduce memory usage. For example, if the mini-batch size changes or the training mode changes, the schedules will need to be recomputed, but memory is also saved by not retaining all cached schedules. The default value is *True*.
- **verbosity_level** (*int*) – Depending on the value, print additional information: 0: Warnings only. (the default value) 1: Show only information that is collected during the first iteration and when a new static schedule is created. 2: Detailed debugging information, possibly showing new information every iteration.
- **enable_double_backprop** (*bool*) – If *True*, enable double-backprop. The default value is *False* (not enabled).

Returns Wrapped `__call__()` method with static chain support.

4.17.2 Basic usage

To enable static graph optimizations, it is only necessary to add the `chainer.static_graph()` decorator to a chain's `__call__()` method. We will now show how the Chainer MNIST example can be modified to use this feature. The modified version with static subgraph optimizations is located at examples/static_graph_optimizations/mnist.

The first step is to import the necessary packages:

Listing 1: train_mnist.py

```
from chainer import static_code
from chainer import static_graph
```

Since the neural network model `MLP` corresponds to a static graph, we can annotate it as a static graph by using the `chainer.static_graph()` decorator on the chain's `__call__()` method. This lets the framework know that that the define-by-run code of the chain always creates the same graph (that is, it always performs the same sequence of computations) each time it is called. We will refer to such a chain as a **static chain** in the documentation.

Listing 2: train_mnist.py

```
# Network definition
class MLP(chainer.Chain):

    """A fully-connected neural network for digit classification.

    """

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            # the size of the inputs to each layer will be inferred
            self.l1 = L.Linear(None, n_units)  # n_in -> n_units
            self.l2 = L.Linear(None, n_units)  # n_units -> n_units
            self.l3 = L.Linear(None, n_out)  # n_units -> n_out

    @static_graph
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

Note: If your model's define-by-run code has any control flow operations that could cause it to potentially call different Chainer functions/links each time it is called, then you cannot use this decorator.

Note: There are currently some restrictions on how variables can be passed into a static chain's `__call__()` method. Refer to the documentation of `chainer.static_graph()` for details.

Recall that the define-by-run code of a static chain's `__call__()` method only actually runs during the first iteration and is then replaced by optimized static schedule code. The current implementation only knows how to do this auto-replacement for calls to Chainer functions and links. Any other code that the user puts in `__call__()` (which we refer to as "side-effect code") will only ever get called once by default, since the define-by-run code is only executed during the first iteration. In order to make sure such "side effect" code actually gets called each iteration, we need to put it inside a function or method decorated by `static_code()`. We expect there will rarely be a need to use side-effect code but for completeness, an example of a model that uses it is available in the `MLPSideEffect` Chain of the static graph MNIST example.

In this example, we only need to use `chainer.static_graph()` on the model chain, since the whole model is static. However, in more general dynamic models, each of the largest static subgraphs (which should each be written as a chain) should also use `chainer.static_graph()`.

Note: Nested application of `chainer.static_graph()` is not allowed. That is, if a `chainer.static_graph()`-decorated chain calls another chains, only the outermost chain should use the decorator.

4.17.3 Calling a static chain multiple times in the same iteration

In a general dynamic graph network, it is not possible to know in advance how many times a static chain will be called in any particular iteration. Note that during training, it is necessary to maintain separate internal state (such as intermediate activations) for each of these calls so that the gradients can be computed in the backward pass. So, although the layer functions of the static schedule will be identical each time the same static chain is called, any internal state must be distinct. It is also possible that a static chain could be called multiple times with inputs of different shapes and/or types during the same iteration. To avoid confuction, "static schedule" will refer to both the functions and any corresponding internal state such as activations.

If backpropagation mode is disabled (`chainer.config.enable_backprop` is `False`), it is safe for the implementation to simply compute a static schedule for the first call and reuse it for subsequent calls, provided that the cached schedule is compatible with the input shapes/types. However, during training, it is necessary to maintain distinct internal state for each call in order to compute the gradients for the backward pass, which prevents us from reusing the same static schedule for each of the multiple calls of a static chain in an iteration.

The current implementation handles this issues as follows. A cache of static schedules, which is intially empty, is associated with each static chain. The size of this cache will be equal to the maximum number of times that the static chain has been called in any previous iteration, and the cache is reset whenever certain chain configuration flags change, such as training mode and backpropagation model. At the start of a given iteration, all cached schedules are available for use and the number of available schedules is decremented each time the static chain is called. If the chain is called when the cache is size zero, then its define-by-run code will execute to create a new schedule cache.

In order for such an implementation to work, each static chain must be notified when the forward pass has ended (or when the forward pass is started) so that all cached schedules can be made available for use again. In the current implementation, this is accomplished by calling the `backward()` method on a loss variable in the model. This is expected to handle the typical use cases. However, in some models it may be necessary to perform multiple forward passes before calling `backward()`. In such a case, to signel to a static chain that the forward pass (and the iteration) has ended, call `my_chain.schedule_manager.end_forward()`. The *schedule_manager* attribute of a static chain is an instance of a class called `StaticScheduleFunction` that will be available after the chain has been called.

4.17.4 Effects on model debugging

Note that since the code in the static chain's `__call__()` only runs during the first iteration, you will only be able to debug this code as define-by-run during the first iteration. It is assumed that if the chain is actually is static, any problems in its define-by-run code should be apparent during the first iteration and it should not be (as) necessary to debug this code in later iterations. However, this feature does provide some functionality to help with debugging. For example, it is possible to obtain and inspect the current static schedules. It is also possible to directly step through the code of the static schedule if you wish (by debugging the `forward()` method of `StaticScheduleFunction` in *`static_graph`*).

4.17.5 Disabling the static subgraph optimization

It is possible to turn off the static subgraph optimization feature by setting the `chainer.config.use_static_graph` to `False`. If set to `False`, the *`chainer.static_graph()`* decorator will simply call the wrapped function without any further side effects.

4.17.6 Limitations and future work

- Optimization switches to let the user select the trade-off between runtime performance and memory usage: The current implementation achieves its speedups mainly by reducing the amount of Python code that needs to run, but does not yet implement advanced optimizations for memory usage or runtime performance. Ideally, the user should be able to adjust performance tuning parameters to control the trade-off between memory consumption and runtime performance.
- Incompatibility with GRU and LSTM links: This feature requires that all input variables to a chain need to explicitly appear in the arguments to the chain's `__call__()` method. However, the GRU and LSTM links with state maintain variable attributes of the chain for the RNN state variables. Design changes to support such links and/or modifications to these links are being considered. These links may still be used with the current implementation, as long as the corresponding RNN is unrolled inside of a static chain. For an example of this, see the modified ptb example at examples/static_graph_optimizations/ptb
- Memory usage: The current implementation caches all static schedules which can lead to high memory usage in some cases. For example, separate schedules are created when the training mode or mini-batch size changes.
- Advanced graph optimizations: Advanced optimizations such as fusion of operations is not yet implemented.
- Constraints on arguments to a static chain: The current version requires that all input variables used inside `__call__()` of a static chain must either appear in the arguments of this method or be defined in the define-by-run code. Furthermore, any variables that appear in the arguments list must appear by themselves or be contained inside a list or tuple. Arbitrary levels of nesting are allowed.
- Model export: In the case where the complete computation graph for the model is static, it should be possible in principle to export the static schedule in a format that can be run on other platforms and languages. One of the other original motivations for this feature was to support exporting static Chainer models to run on C/C++ and/or optimize the static schedule execution code in Cython/C/C++. However, it seems that ONNX is now

fulfilling this purpose and there is a separate ONNX exporter already in development for Chainer. Perhaps these two features can be merged at some point in the future.

- Double-backward support: This feature was designed to support double-backward (gradient of gradient) but it has not been tested.
- ChainerX is not supported. If you have code written using this feature but would like to run the model with ChainerX, please set the `chainer.config.use_static_graph` configuration to `False`. The code should then work without any additional changes.

4.17.7 Examples

For additional examples that use this feature, refer to the examples in examples/static_graph_optimizations.

4.18 Static Subgraph Optimizations: Design Notes

This documentation is intended provide information on the architecture and design of the static subgraph optimizations feature for those who are interested in contributing to its development. This documentation also describes how existing Chainer functions can be modified to run more efficiently when static subgraph optimizations are enabled.

4.18.1 Overview of dynamic and static graph frameworks

Existing deep learning frameworks can roughly be classified as either a “static graph” or “dynamic graph” framework. In a static graph framework, which we also call “define-and-run”, the computation graph is defined before the model is run. This implies that the same neural network model will be used each iteration without modifications, hence the name “static.” This allows various graph optimizations to potentially be performed to improve the runtime performance and/or reduce memory usage. The optimized code for the computation graph is then used when the model is run.

However, in a “dynamic graph” (also called “define-by-run”) framework such as Chainer, the computation graph is not defined before the model is run. Rather, it is constructed incrementally and automatically by the framework as the computations of the forward pass are executed. In Chainer, the user writes code to perform the computations of the forward pass in terms of Chainer functions, which have an API similar to an array library like NumPy. As these functions execute, the computation graph is incrementally built so that it will be available after the last function in the forward pass has been called. This has some advantages, such as allowing easier debugging compared to a static graph framework, since the user can step through the computations of the forward pass in a debugger. Define-by-run also provides the flexibility to include control flow operations so that a modified or even completely different graph can be constructed each iteration. Unfortunately, this flexibility also tends to make dynamic graph frameworks slower than static graph frameworks. For example, in Chainer there is a performance penalty involved in dynamically constructing the graph each iteration, since it involves creating many objects; each function call creates a new *FunctionNode* object as well as creating new *VariableNode* and array memory allocation for each output of the function. There are also various dynamic type checks and graph traversal that need to be performed, adding to the runtime overhead. Further, we cannot perform some optimizations such as function/kernel fusion and in-place operations.

4.18.2 Static subgraph optimizations feature

This feature is motivated by the observation that typical deep neural networks correspond to a static computation graph and that even those that correspond to a dynamic graph are typically mostly static. By “mostly static”, we mean that the largest static subgraphs each tend to contain many function nodes (that is, layers) so that the total number of function nodes in the graph tends to be much larger than the total number of largest static subgraphs. If the graph is at least mostly static, then a naive implementation of define-by-run will result in a large amount of redundant operations being performed each iteration to rebuild exactly the same subgraphs, perform the same dynamic type-checking operations, etc., which can sometimes be slow in Python; it will also result in lost opportunities to perform potential graph optimizations. A key assumption motivating this feature is that the main performance bottlenecks tend to occur inside the largest static subgraphs. So, if we can optimize these static subgraphs, it might be fine for any remaining framework code to remain implemented in pure Python. Although such Python code would be slow, it could have negligible runtime overhead.

The solution proposed by this feature is to retain the existing define-by-run style for specifying the model, but to also optionally allow the user to annotate the largest static subgraphs in a model. These "static graph" annotations will then allow the framework to automatically replace the define-by-run code of the static subgraphs with more performance-optimized code. The define-by-run code will still execute during the first iteration, to retain ease of debugging. However, as this code executes, a trace of the needed computations is also collected so that optimized static schedules can be generated for the annotated static subgraphs. Then, starting from the second iteration, this optimized code will automatically be run in place of the original define-by-run code. Note that in the common case in which the whole model is static, the user only needs to add a single "static graph" annotation and their code will then run with the performance of a static graph framework, while still supporting the define-by-run coding style.

The benefit of annotating the static subgraphs in the model is that it allows the define-by-run code to be replaced with an optimized static schedule, which can then potentially support a user-controllable trade-off between runtime performance and memory usage. This is possible because having the full computation graph available enables various optimizations that cannot safely or automatically be performed in define-by-run. Examples (which we have not yet implemented; contributions from the open source community are welcomed) include sub-linear memory usage [1], exploiting graph parallelism, operator fusion, and in-place optimizations.

The current implementation achieves its speedup by retaining only the code that is actually needed to compute the forward pass, backward pass, and so on. This allows us to remove most of the Python interpreter overhead because the Python code that performs dynamic operations such as allocating *FunctionNode* and *Variable* objects, checking types, and traversing the backward graph is not included in the optimized static schedule code.

4.18.3 Adding support to existing functions

Most functions and links will not need to be modified at all in order to support this feature, since the framework code will attempt to auto-wrap them inside a *@static_code*-decorated function. However, some functions might see a performance benefit if static graph support is added manually, since it may result in less redundant code being included in the static schedule. For example, any dynamic checking code that will return the same result every iteration does not need to be included in the static schedule.

An existing function (that is, a subclass of *FunctionNode*) can be modified to support static graph optimizations as follows. The basic idea is to wrap any code that needs to be called each iteration inside a method that is decorated with `@static_code`. Note that code that should only run once, such as initializing parameters, should not be wrapped.

It is also necessary to set the `_supports_static_optimizations = True` class attribute. Note that this attribute is `False` by default in `FunctionNode`.

Since the function is part of a static graph, any parameters and output arrays should ideally be statically allocated during the first iteration (while the define-by-run code is executing) and then reused starting from the second iteration. The `@static_code`-decorated functions that are called each iteration will perform the various deep learning computations, writing results in-place into these static arrays. Since the results are written in-place, there is no need for an *@static_code*-decorated function to explicitly return a result. Rather, any results arrays should be passed as inputs along with any other input arguments to the function. However, it also is allowed to return dynamically allocated arrays so that existing Chainer functions can be easily supported. The following code shows the typical pattern for performing the forward computations in a *FunctionNode*:

```
@static_code
def static_forward(self, inputs, outputs):
    # This function will get
```

(continues on next page)

(continued from previous page)

```
included in the static
        # schedule and called each iteration.
        # Any input arrays must be passed in a list
        # to the `inputs` keyword argument.
        x = inputs[0]
        # Any output arrays must be passed in a list
        # to the `outputs` keyword argument, and must
        # have already been initialized to the required
        # shape. Results are written in-place into output
        # arrays.
        y = outputs[0]

        # Read from x, write results into y in-place.
        # Don't forget to zero y if necessary.
        y *= 0.0 # (if necessary)
        y[:] = 3.0*x # for example

    def forward(self, inputs):
        # Initialization/type checking code.
        # (only gets called once, during first iteration)
        type_check_blah(inputs)

        # Allocate output array. Note that since this line
        # is not wrapped using @static_code, it
        # will only ever get called once, during the first
        # iteration.
        y = xp.empty(y_shape).astype(x.dtype)

        # Call static function
        # (it will get called every iteration from optimized schedule)
        self.static_forward(inputs=[x], outputs=[y])
        return y,
```

It should not be necessary to modify the *backward()* implementation. As of Chainer v3 when double-backward (i.e., grad of grad) support was added, the `backward()` method of *FunctionNode* actually calls the *forward()* method of other *FunctionNode‘s, and so it is only necessary that the ‘forward()* functions be wrapped.

For an example of how to add support to an existing function, see the `Linear` function.

4.18.4 Adding support to existing links

Most existing links will work as-is and do not need to be modified. However, if a link needs to perform computations each iteration that are performed in code other than calling chainer functions, this code will need to be manually placed in a @*static_code*-decorated function or method of the link.

If a link performs different computations depending on the training mode but is otherwise static, then it does not need to be modified.

4.18.5 Reference

[1] Training deep nets with sublinear memory cost

4.19 Caffe Model Support

Caffe is a popular framework maintained by BVLC at UC Berkeley. It is widely used by computer vision communities, and aims at fast computation and easy usage without any programming. The BVLC team provides trained reference models in their Model Zoo, which can reduce training time required for a new task.

4.19.1 Import

Chainer can import the reference models and emulate the network by `Link` implementations. This functionality is provided by the `chainer.links.caffe.CaffeFunction` class.

`chainer.links.caffe.CaffeFunction`	Caffe emulator based on the model file of Caffe.

4.19.2 Export

Chainer can export a model from `Link`.

`chainer.exporters.caffe.export`	(Experimental) Export a computational graph as Caffe format.

chainer.exporters.caffe.export

`chainer.exporters.caffe.`**`export`**(*model*, *args*, *directory=None*, *export_params=True*, *graph_name='Graph'*)

(Experimental) Export a computational graph as Caffe format.

Parameters

- **model** (`Chain`) – The model object you want to export in Caffe format. It should have `__call__()` method because the second argument `args` is directly given to the model by the `()` accessor.
- **args** (*list of ~chainer.Variable*) – The arguments which are given to the model directly.
- **directory** (*str*) – The directory used for saving the resulting Caffe model. If None, nothing is saved to the disk.
- **export_params** (*bool*) – If True, this function exports all the parameters included in the given model at the same time. If False, the exported Caffe model doesn't include any parameter values.
- **graph_name** (*str*) – A string to be used for the `name` field of the graph in the exported Caffe model.

Note: Currently, this function supports networks that created by following layer functions.

- `linear()`
- `convolution_2d()`
- `deconvolution_2d()`
- `max_pooling_2d()`

- *average_pooling_2d()*
- *batch_normalization()*
- *local_response_normalization()*
- *relu()*
- *leaky_relu()*
- *concat()*
- *softmax()*
- *reshape()*
- *add()*

This function can export at least following networks.

- GoogLeNet
- ResNet
- VGG

And, this function use testing (evaluation) mode.

Example

```
>>> from chainer.exporters import caffe
>>>
>>> class Model(chainer.Chain):
...     def __init__(self):
...         super(Model, self).__init__()
...         with self.init_scope():
...             self.l1 = L.Convolution2D(None, 1, 1, 1, 0)
...             self.b2 = L.BatchNormalization(1)
...             self.l3 = L.Linear(None, 1)
...
...     def __call__(self, x):
...         h = F.relu(self.l1(x))
...         h = self.b2(h)
...         return self.l3(h)
...
>>> x = chainer.Variable(np.zeros((1, 10, 10, 10), np.float32))
>>> caffe.export(Model(), [x], None, True, 'test')
```

4.20 Assertion and Testing

Chainer provides some facilities to make debugging easy.

4.20.1 Type checking utilities

FunctionNode uses a systematic type checking of the *chainer.utils.type_check* module. It enables users to easily find bugs of forward and backward implementations. You can find examples of type checking in some function implementations.

chainer.utils.type_check.Expr	Abstract syntax tree of an expression.
chainer.utils.type_check.eval	
chainer.utils.type_check.expect	Evaluates and tests all given expressions.
chainer.utils.type_check.TypeInfo	Type information of an input/gradient array.
chainer.utils.type_check.TypeInfoTuple	Type information of input/gradient tuples.
chainer.utils.type_check.Variable	

chainer.utils.type_check.Expr

class chainer.utils.type_check.**Expr**(*priority*)

Abstract syntax tree of an expression.

It represents an abstract syntax tree, and isn't a value. You can get its actual value with *eval()* function, and get syntax representation with the `__str__()` method. Each comparison operator (e.g. ==) generates a new *Expr* object which represents the result of comparison between two expressions.

Example

Let `x` and `y` be instances of *Expr*, then

```
>>> x = Variable(1, 'x')
>>> y = Variable(1, 'y')
>>> c = (x == y)
```

is also an instance of *Expr*. To evaluate and get its value, call *eval()* method:

```
>>> c.eval()
True
```

Call `str` function to get a representation of the original equation:

```
>>> str(c)
'x == y'
```

You can actually compare an expression with a value:

```
>>> (x == 1).eval()
True
```

Note that you can't use boolean operators such as `and`, as they try to cast expressions to boolean values:

```
>>> z = Variable(1, 'z')
>>> x == y and y == z  # raises an error
Traceback (most recent call last):
RuntimeError: Don't convert Expr to bool. Please call Expr.eval method to␣
↪evaluate expression.
```

Methods

__call__(**args*)

Call self as a function.

__getitem__(*key*)

eval()

Evaluates the tree to get actual value.

Behavior of this function depends on an implementation class. For example, a binary operator + calls the `__add__` function with the two results of `eval()` function.

__eq__(*y*)

__ne__(*y*)

__lt__(*y*)

__le__(*y*)

__gt__(*y*)

__ge__(*y*)

__nonzero__()

__bool__()

__neg__()

__add__(*y*)

__radd__(*y*)

__sub__(*y*)

__rsub__(*y*)

__mul__(*y*)

__rmul__(*y*)

__truediv__(*y*)

__rtruediv__(*y*)

__floordiv__(*y*)

__rfloordiv__(*y*)

__pow__(*y*)

chainer.utils.type_check.eval

chainer.utils.type_check.**eval**(*exp*)

chainer.utils.type_check.expect

chainer.utils.type_check.**expect**(**bool_exprs*)

Evaluates and tests all given expressions.

This function evaluates given boolean expressions in order. When at least one expression is evaluated as `False`, that means the given condition is not satisfied. You can check conditions with this function.

Parameters **bool_exprs** (*tuple of Bool expressions*) – Bool expressions you want to evaluate.

chainer.utils.type_check.TypeInfo

class chainer.utils.type_check.**TypeInfo**(*shape*, *dtype*)

Type information of an input/gradient array.

It contains type information of an array, such as the shape of array and the number of dimensions. This information is independent of CPU or GPU array.

Methods

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

size

chainer.utils.type_check.TypeInfoTuple

class chainer.utils.type_check.**TypeInfoTuple**

Type information of input/gradient tuples.

It is a sub-class of tuple containing *TypeInfo*. The i-th element of this object contains type information of the i-th input/gradient data. As each element is *Expr*, you can easily check its validity.

Methods

__getitem__()
Return self[key].

__len__()
Return len(self).

__iter__()
Implement iter(self).

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

size()
Returns an expression representing its length.

Returns An expression object representing length of the tuple.

Return type *Expr*

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

__add__()
Return self+value.

__mul__()
Return self*value.n

__rmul__()
Return self*value.

chainer.utils.type_check.Variable

class chainer.utils.type_check.**Variable**(*value*, *name*)

Methods

__call__(**args*)
Call self as a function.

__getitem__(*key*)

eval()
Evaluates the tree to get actual value.

Behavior of this function depends on an implementation class. For example, a binary operator + calls the __add__ function with the two results of `eval()` function.

__eq__(*y*)

__ne__(*y*)

__lt__(*y*)

__le__(*y*)

__gt__(*y*)

__ge__(*y*)

__nonzero__()

__bool__()

__neg__()

__add__(*y*)

__radd__(*y*)

__sub__(*y*)

__rsub__(*y*)

__mul__(*y*)

__rmul__(*y*)

__truediv__(*y*)

__rtruediv__(*y*)

__floordiv__(*y*)

__rfloordiv__(*y*)

__pow__(*y*)

4.20.2 Gradient checking utilities

Most function implementations are numerically tested by *gradient checking*. This method computes numerical gradients of forward routines and compares their results with the corresponding backward routines. It enables us to make the source of issues clear when we hit an error of gradient computations. The `chainer.gradient_check` module makes it easy to implement the gradient checking.

`chainer.gradient_check.check_backward`	Test backward procedure of a given function.
`chainer.gradient_check.check_double_backward`	Test twice differentiation of a given procedure.
`chainer.gradient_check.numerical_grad`	Computes numerical gradient by finite differences.

chainer.gradient_check.check_backward

`chainer.gradient_check.`**`check_backward`**(*func, x_data, y_grad, params=(), eps=0.001, atol=1e-05, rtol=0.0001, no_grads=None, dtype=None, detect_nondifferentiable=False*)

Test backward procedure of a given function.

This function automatically checks the backward-process of a given function to ensure that the computed gradients are approximately correct. For example, assuming you've defined a `FunctionNode` class `MyFunc`, that takes two arguments and returns one value, you can wrap it in a ordinary function and check its gradient computations as follows:

```
def func(xs):
    y, = MyFunc().apply(xs)
    return y

x1_data = xp.array(...)
```

(continues on next page)

(continued from previous page)

```
x2_data = xp.array(...)
gy_data = xp.array(...)
check_backward(func, (x1_data, x2_data), gy_data)
```

This function creates *Variable* objects with `x_data` and calls `func` with the *Variable*s to get its result as *Variable*. Then, it sets `y_grad` array to `grad` attribute of the result and calls `backward` method to get gradients of the inputs. To check correctness of the gradients, the function calls *numerical_grad()* to calculate numerically the gradients and compares the types of gradients with *chainer.testing.assert_allclose()*.

To reduce computational time, it uses directional derivative along a random vector. A function $g : \mathbb{R} \to \mathbb{R}^n$ is defined as $g(\delta) = f(x + \delta r)$, where $\delta \in \mathbb{R}$, $r \in \mathbb{R}^n$ is a random vector and f is a function which you want to test. Its gradient is

$$g'(\delta) = f'(x + \delta r) \cdot r.$$

Therefore, $g'(0) = f'(x) \cdot r$. So we can check the correctness of back propagation of f indirectly by comparing this equation with the gradient of g numerically calculated and that of f computed by backprop. If r is chosen from uniform distribution, we can conclude with high probability that the gradient of f itself is correct.

If the function is non-differentiable with respect to some input objects, we can check its backprop to such objects by `no_grads` argument. `gradient_check` computes numerical backward to inputs that correspond to `False` in `no_grads`. It also asserts that the backprop leaves gradients `None` for inputs that correspond to `True` in `no_grads`. The default of `no_grads` argument is the tuple of truth values whether input objects (`x1_data` or/and `x2_data` in this example) represent integer variables.

You can simplify a test when `MyFunc` gets only one argument:

```
check_backward(func, x1_data, gy_data)
```

If `MyFunc` is a loss function which returns a zero-dimensional array, pass `None` to `gy_data`. In this case, it sets `1` to `grad` attribute of the result:

```
check_backward(my_loss_func,
               (x1_data, x2_data), None)
```

If `MyFunc` returns multiple outputs, pass all gradients for outputs as a tuple:

```
gy1_data = xp.array(...)
gy2_data = xp.array(...)
check_backward(func, x1_data, (gy1_data, gy2_data))
```

You can also test a *Link*. To check gradients of parameters of the link, set a tuple of the parameters to `params` arguments:

```
check_backward(my_link, (x1_data, x2_data), gy_data,
               (my_link.W, my_link.b))
```

Note that `params` are not `ndarrays`, but `Variabless`.

Function objects are acceptable as `func` argument:

```
check_backward(lambda x1, x2: f(x1, x2),
               (x1_data, x2_data), gy_data)
```

Note: func is called many times to get numerical gradients for all inputs. This function doesn't work correctly when func behaves randomly as it gets different gradients.

Parameters

- **func** (*callable*) – A function which gets *Variable*s and returns *Variable*s. func must returns a tuple of *Variable*s or one *Variable*. You can use a *Function*, *FunctionNode* or a *Link* object or any other function satisfying the condition.
- **x_data** (ndarray *or tuple of ndarrays*) – A set of ndarrays to be passed to func. If x_data is one ndarray object, it is treated as (x_data,).
- **y_grad** (ndarray *or tuple of ndarrays or* None) – A set of ndarrays representing gradients of return-values of func. If y_grad is one ndarray object, it is treated as (y_grad,). If func is a loss-function, y_grad should be set to None.
- **params** (Variable *or tuple of ~chainder.Variable*) – A set of *Variable*s whose gradients are checked. When func is a *Link* object, set its parameters as params. If params is one *Variable* object, it is treated as (params,).
- **eps** (*float*) – Epsilon value to be passed to *numerical_grad()*.
- **atol** (*float*) – Absolute tolerance to be passed to *chainer.testing.assert_allclose()*.
- **rtol** (*float*) – Relative tolerance to be passed to *chainer.testing.assert_allclose()*.
- **no_grads** (*list of bool*) – Flag to skip variable for gradient assertion. It should be same length as x_data.
- **dtype** (*dtype*) – x_data, y_grad and params are casted to this dtype when calculating numerical gradients. Only float types and None are allowed.
- **detect_nondifferentiable** (*bool*) – If True, check for non-differentiable inputs is enabled. If func is non-differentiable at x_data, check_backward raises NondifferentiableError.

See also:

numerical_grad()

chainer.gradient_check.check_double_backward

chainer.gradient_check.**check_double_backward**(*func, x_data, y_grad, x_grad_grad, params=(), params_grad_grad=(), eps=0.001, atol=0.0001, rtol=0.001, no_grads=None, dtype=None, detect_nondifferentiable=False*)

Test twice differentiation of a given procedure.

This function automatically checks if the backward procedure of func is correctly implemented for further differentiation. It first computes the gradient of func w.r.t. its inputs in the same way as *check_backward()*. This function then further invokes the backward procedure against the gradient variables, starting from the initial gradient given by x_grad_grad. It also computes the second gradient using *numerical_grad()*. The resulting gradients are compared to confirm if the second-order gradients are approximately correct.

Note that this function **DOES NOT** check if the first-order differentiation is correct; the numerical gradient assumes that the first-order gradient given by the usual *chainer.Variable.backward()* is correct. The implementation of each differentiable function should be tested by *check_backward()* first, and then should be tested by this function if neccessary.

For the details of the arguments, see *check_backward()*. The additional arguments `x_grad_grad` and `params_grad_grad` are (tuples of) *Variable* (s) that include the initial gradient corresponding to the first-order gradient of each input and parameter. Note that the default error tolerance `atol` and `rtol` are slightly larger than those of *check_backward()* because the numerical gradients of the second order differentiation are less accurate than those of the first order gradients.

chainer.gradient_check.numerical_grad

`chainer.gradient_check.`**`numerical_grad`**(*f*, *inputs*, *grad_outputs*, *eps=0.001*, *detect_nondifferentiable=False*, *diff_atol=0*, *diff_rtol=0.01*, *center_outputs=None*)

Computes numerical gradient by finite differences.

This function is used to implement gradient check. For usage example, see unit tests of *chainer.functions*.

By default, `numerical_grad` computes the gradient to the first order of `eps`.

Parameters

- **f** (*callable*) – Python function with no arguments that runs forward computation and returns the result.
- **inputs** (*tuple of arrays*) – Tuple of arrays that should be treated as inputs. Each element of them is slightly modified to realize numerical gradient by finite differences.
- **grad_outputs** (*tuple of arrays or scalars*) – Tuple of arrays or scalars that are treated as output gradients.
- **eps** (*float*) – Epsilon value of finite differences.
- **detect_nondifferentiable** (*bool*) – `False` by default. If `True`, `numerical_grad` checks whether `f` is differentiable at `inputs`. It requires evaluation of `f` at 5 points instead of 2. As a side effect, the accuracy of numerical gradient will be increased to the third order of `eps`. If it turns out that `f` is non-differentiable at `input`, `numerical_grad` raises `NondifferentiableError`.
- **diff_atol** (*float*) – Absolute tolerance of fitting error of non-differentiable point detection.
- **diff_rtol** (*float*) – Tolerance of fitting error of non-differentiable point detection relative to the output values of `f`.
- **center_outputs** (*tuple of arrays or None*) – Only used if `detect_nondifferentiable` is `True`. If specified, these arrays are used as the outputs of `f` at `inputs`. Otherwise, it is calculated. It can be used to reduce the computation if these arrays are already calculated before calling `numerical_grad`.

Returns Numerical gradient arrays corresponding to `inputs`.

Return type tuple

4.20.3 Standard Assertions

The assertions have same names as NumPy's ones. The difference from NumPy is that they can accept both `numpy.ndarray` and `cupy.ndarray`.

chainer.testing.assert_allclose	Asserts if some corresponding element of x and y differs too much.
chainer.testing.assert_warns	

chainer.testing.assert_allclose

`chainer.testing.`**`assert_allclose`**(*x*, *y*, *atol=1e-05*, *rtol=0.0001*, *verbose=True*)

Asserts if some corresponding element of x and y differs too much.

This function can handle both CPU and GPU arrays simultaneously.

Parameters

- **x** – Left-hand-side array.
- **y** – Right-hand-side array.
- **atol** (*float*) – Absolute tolerance.
- **rtol** (*float*) – Relative tolerance.
- **verbose** (*bool*) – If `True`, it outputs verbose messages on error.

chainer.testing.assert_warns

`chainer.testing.`**`assert_warns`**(*expected*)

4.20.4 Function testing utilities

Utilities for testing functions.

chainer.testing.FunctionTestCase	A base class for function test cases.
chainer.testing.unary_math_function_unittest	Decorator for testing unary mathematical Chainer functions.

chainer.testing.FunctionTestCase

class `chainer.testing.`**`FunctionTestCase`**(**args*, ***kwargs*)

A base class for function test cases.

Function test cases can inherit from this class to define a set of function tests.

Required methods

Each concrete class must at least override the following three methods.

forward(self, inputs, device) Implements the target forward function. `inputs` is a tuple of *Variable*s. This method is expected to return the output *Variable*s with the same array types as the inputs. `device` is the device corresponding to the input arrays.

forward_expected(self, inputs) Implements the expectation of the target forward function. `inputs` is a tuple of `numpy.ndarray`s. This method is expected to return the output `numpy.ndarray`s.

generate_inputs(self) Returns a tuple of input arrays of type `numpy.ndarray`.

Optional methods

Additionally the concrete class can override the following methods.

before_test(self, test_name) A callback method called before each test. Typically a skip logic is implemented by conditionally raising `unittest.SkipTest`. `test_name` is one of `'test_forward'`, `'test_backward'`, and `'test_double_backward'`.

generate_grad_outputs(self, outputs_template) Returns a tuple of output gradient arrays of type `numpy.ndarray` or `None` for omitted the gradients. `outputs_template` is a tuple of template arrays. The returned arrays are expected to have the same shapes and dtypes as the template arrays.

generate_grad_grad_inputs(self, inputs_template) Returns a tuple of the second order input gradient arrays of type `numpy.ndarray` or `None` for omitted gradients. `input_template` is a tuple of template arrays. The returned arrays are expected to have the same shapes and dtypes as the template arrays.

check_forward_outputs(self, outputs, expected_outputs) Implements check logic of forward outputs. Typically additional check can be done after calling `super().check_forward_outputs`. `outputs` and `expected_outputs` are tuples of arrays. In case the check fails, `FunctionTestError` should be raised.

Configurable attributes

The concrete class can override the following attributes to control the behavior of the tests.

skip_forward_test (bool): Whether to skip forward computation test. `False` by default.

skip_backward_test (bool): Whether to skip backward computation test. `False` by default.

skip_double_backward_test (bool): Whether to skip double-backward computation test. `False` by default.

dodge_nondifferentiable (bool): Enable non-differentiable point detection in numerical gradient calculation. If the inputs returned by `generate_inputs` turns out to be a non-differentiable point, the test will repeatedly resample inputs until a differentiable point will be finally sampled. `False` by default.

numerical_grad_dtype (dtype): Input arrays are casted to this dtype when calculating the numerical gradients. It is `float64` by default, no matter what the original input dtypes were, to maximize precision.

contiguous (None or 'C'): Specifies the contiguousness of incoming arrays (i.e. inputs, output gradients, and the second order input gradients). If `None`, the arrays will be non-contiguous as long as possible. If `'C'`, the arrays will be C-contiguous. `None` by default.

Passive attributes

These attributes are automatically set.

test_name (str): The name of the test being run. It is one of `'test_forward'`, `'test_backward'`, and `'test_double_backward'`.

backend_config (BackendConfig): The backend configuration.

Note: This class assumes *chainer.testing.inject_backend_tests()* is used together. See the example below.

Example

```
@chainer.testing.inject_backend_tests(
    None,
    [
        {},  # CPU
        {'use_cuda': True},  # GPU
    ])
class TestReLU(chainer.testing.FunctionTestCase):

    # ReLU function has a non-differentiable point around zero, so
    # dodge_nondifferentiable should be set to True.
    dodge_nondifferentiable = True

    def generate_inputs(self):
        x = numpy.random.uniform(-1, 1, (2, 3)).astype(numpy.float32)
        return x,

    def forward(self, inputs, device):
        x, = inputs
        return F.relu(x),

    def forward_expected(self, inputs):
        x, = inputs
        expected = x.copy()
        expected[expected < 0] = 0
        return expected,
```

See also:

LinkTestCase

Methods

__call__(**args*, ***kwds*)
Call self as a function.

addCleanup(*function*, **args*, ***kwargs*)
Add a function, with arguments, to be called when the test is completed. Functions added are called on a LIFO basis and are called after tearDown on test failure or success.

Cleanup items are called even if setUp fails (unlike tearDown).

addTypeEqualityFunc(*typeobj*, *function*)
Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Parameters

- **typeobj** – The data type to call this function on when both values are of the same type in assertEqual().
- **function** – The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

assertAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertAlmostEquals (***kwargs*)

assertCountEqual (*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset (*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertDictEqual (*d1*, *d2*, *msg=None*)

assertEqual (*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEquals (***kwargs*)

assertFalse (*expr*, *msg=None*)
Check that the expression is false.

assertGreater (*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance (*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone (*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone (*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess (*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual (*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs (*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual (*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

Objects that are equal automatically fail.

assertNotAlmostEquals (***kwargs*)

assertNotEqual (*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotEquals (***kwargs*)

assertNotIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance (*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex (*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNotRegexpMatches(***kwargs*)

assertRaises(*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRaisesRegexp(***kwargs*)

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertRegexpMatches(***kwargs*)

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

assert_(***kwargs*)

before_test(*test_name*)

check_forward_outputs(*outputs*, *expected_outputs*)

countTestCases()

debug()
Run the test without collecting errors in a TestResult

defaultTestResult()

doCleanups()
Execute all cleanup functions. Normally called for you after tearDown.

fail(*msg=None*)
Fail immediately, with the given message.

failIf(***kwargs*)

failIfAlmostEqual(***kwargs*)

failIfEqual(***kwargs*)

failUnless(***kwargs*)

failUnlessAlmostEqual(***kwargs*)

failUnlessEqual(***kwargs*)

failUnlessRaises(***kwargs*)

forward(*inputs*, *device*)

forward_expected(*inputs*)

generate_grad_grad_inputs(*inputs_template*)

generate_grad_outputs(*outputs_template*)

generate_inputs()

id()

run(*result=None*)

run_test_backward(*backend_config*)

run_test_double_backward(*backend_config*)

run_test_forward(*backend_config*)

setUp()
Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

subTest(*msg=<object object>*, ***params*)
Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

tearDown()
Hook method for deconstructing the test fixture after testing it.

classmethod tearDownClass()
Hook method for deconstructing the class fixture after running all tests in the class.

test_backward(*backend_config*)
Tests backward computation.

test_double_backward(*backend_config*)
Tests double-backward computation.

test_forward(*backend_config*)
Tests forward computation.

__eq__(*other*)
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

backend_config = None

check_backward_options = None

check_double_backward_options = None

check_forward_options = None

contiguous = None

dodge_nondifferentiable = False

longMessage = True

maxDiff = 640

skip_backward_test = False

skip_double_backward_test = False

skip_forward_test = False

chainer.testing.unary_math_function_unittest

`chainer.testing.unary_math_function_unittest(func, func_expected=None, label_expected=None, make_data=None, is_linear=None, forward_options=None, backward_options=None, double_backward_options=None)`

Decorator for testing unary mathematical Chainer functions.

This decorator makes test classes test unary mathematical Chainer functions. Tested are forward and backward, including double backward, computations on CPU and GPU across parameterized `shape` and `dtype`.

Parameters

- **func** (*function or* `Function`) – Chainer function to be tested by the decorated test class. Taking `Function` is for backward compatibility.
- **func_expected** – Function used to provide expected values for testing forward computation. If not given, a corresponsing numpy function for `func` is implicitly picked up by its name.
- **label_expected** (*string*) – String used to test labels of Chainer functions. If not given, the name of `func` is implicitly used.
- **make_data** – Function to customize input and gradient data used in the tests. It takes `shape` and `dtype` as its arguments, and returns a tuple of input, gradient and double gradient data. By default, uniform destribution ranged `[-1, 1]` is used for all of them.
- **is_linear** – Tells the decorator that `func` is a linear function so that it wraps `func` as a non-linear function to perform double backward test. This argument is left for backward compatibility. Linear functions can be tested by default without specifying `is_linear` in Chainer v5 or later.
- **forward_options** (*dict*) – Options to be specified as an argument of `chainer.testing.assert_allclose()` function. If not given, preset tolerance values are automatically selected.
- **backward_options** (*dict*) – Options to be specified as an argument of `chainer.gradient_check.check_backward()` function. If not given, preset tolerance values are automatically selected depending on `dtype`.
- **double_backward_options** (*dict*) – Options to be specified as an argument of `chainer.gradient_check.check_double_backward()` function. If not given, preset tolerance values are automatically selected depending on `dtype`.

The decorated test class tests forward, backward and double backward computations on CPU and GPU across the following `parameterize()` ed parameters:

- shape: rank of zero, and rank of more than zero
- dtype: `numpy.float16`, `numpy.float32` and `numpy.float64`

Additionally, it tests the label of the Chainer function.

Chainer functions tested by the test class decorated with the decorator should have the following properties:

- Unary, taking one parameter and returning one value
- `dtype` of input and output are the same
- Elementwise operation for the supplied ndarray

Example

The following code defines a test class that tests *sin()* Chainer function, which takes a parameter with `dtype` of float and returns a value with the same `dtype`.

```
>>> import unittest
>>> from chainer import testing
>>> from chainer import functions as F
>>>
>>> @testing.unary_math_function_unittest(F.sin)
... class TestSin(unittest.TestCase):
...     pass
```

Because the test methods are implicitly injected to `TestSin` class by the decorator, it is enough to place `pass` in the class definition.

To customize test data, `make_data` optional parameter can be used. The following is an example of testing `sqrt` Chainer function, which is tested in positive value domain here instead of the default input.

```
>>> import numpy
>>>
>>> def make_data(shape, dtype):
...     x = numpy.random.uniform(0.1, 1, shape).astype(dtype)
...     gy = numpy.random.uniform(-1, 1, shape).astype(dtype)
...     ggx = numpy.random.uniform(-1, 1, shape).astype(dtype)
...     return x, gy, ggx
...
>>> @testing.unary_math_function_unittest(F.sqrt,
...                                   make_data=make_data)
... class TestSqrt(unittest.TestCase):
...     pass
```

`make_data` function which returns input, gradient and double gradient data generated in proper value domains with given `shape` and `dtype` parameters is defined, then passed to the decorator's `make_data` parameter.

4.20.5 Link testing utilities

Utilities for testing links.

chainer.testing.LinkInitializersTestCase	A base class for link parameter initializer test cases.
chainer.testing.LinkTestCase	A base class for link forward and backward test cases.

chainer.testing.LinkInitializersTestCase

class chainer.testing.**LinkInitializersTestCase**(**args*, ***kwargs*)

A base class for link parameter initializer test cases.

Link test cases can inherit from this class to define a set of link tests for parameter initialization.

Required methods

Each concrete class must at least override the following methods.

generate_params(self) Returns a tuple of initializers-likes. The tuple should contain an initializer-like for each initializer-like argument, i.e. the parameters to the link constructor. These will be passed to `create_link`.

create_link(self, initializers) Returns a link. The link should be initialized with the given initializer-likes `initializers`. `initializers` is a tuple of same length as the number of parameters.

generate_inputs(self) Returns a tuple of input arrays of type `numpy.ndarray`.

forward(self, link, inputs, device) Implements the target forward function. `link` is a link created by `create_link` and `inputs` is a tuple of *Variable*s. This method is expected to return the output *Variable*s with the same array types as the inputs. `device` is the device corresponding to the input arrays. A default implementation is provided for links that only takes the inputs defined in `generate_inputs` (wrapped in *Variable*s) and returns nothing but output *Variable*s in its forward computation.

get_initializers(self) Returns a tuple with the same length as the number of initializers that the constructor of the link accepts. Each element in the tuple is a container itself, listing all initializers-likes that should be tested. Each initializer-like in the tuple is tested one at a time by being passed to `create_link`. When the length of the tuple is greater than one (i.e. if the link accepts multiple initializers), the ones not being tested are replaced by the ones returned by *generate_params*. Initializer-likes returned here should be deterministic since test will invoke them multiple times to test the correctness.

For testing initializer arguments that can be non-initializer values such as `None`, one can use the `InitializerArgument`, defining a pair of the link constructor argument and actual initializer-like used by the link. This method must be implemented if `skip_initializers_test` is `False` in which case the initializers test is executed.

Optional methods

Each concrete class may override the following methods.

before_test(self, test_name) A callback method called before each test. Typically a skip logic is implemented by conditionally raising `unittest.SkipTest`. `test_name` is always of `'test_initializers'`.

Attributes

The concrete class can override the following attributes to control the behavior of the tests.

param_names (list of str): A list of strings with all the names of the parameters that should be tested. E.g. `['gamma', 'beta']` for the batch normalization link. `[]` by default.

contiguous (None or 'C'): Specifies the contiguousness of incoming arrays (i.e. inputs, parameters and gradients. If `None`, the arrays will be non-contiguous as long as possible. If `'C'`, the arrays will be C-contiguous. `None` by default.

Note: This class assumes *chainer.testing.inject_backend_tests()* is used together. See the example below.

Note: When implementing *LinkTestCase* and *LinkInitializersTestCase* to test both forward/backward and initializers, it is often convenient to refactor out common logic in a separate class.

Example

```
@chainer.testing.inject_backend_tests(
  None,
  [
      {},  # CPU
      {'use_cuda': True},  # GPU
  ])
class TestLinear(chainer.testing.LinkInitializersTestCase):

    param_names = ['W', 'b']

    def generate_params(self):
        initialW = numpy.random.uniform(
            -1, 1, (3, 2)).astype(numpy.float32)
        initial_bias = numpy.random.uniform(
            -1, 1, (3,)).astype(numpy.float32)
        return initialW, initial_bias

    def generate_inputs(self):
        x = numpy.random.uniform(
            -1, 1, (1, 2)).astype(numpy.float32)
        return x,

    def create_link(self, initializers):
        initialW, initial_bias = initializers
        link = chainer.links.Linear(
            2, 3, initialW=initialW, initial_bias=initial_bias)
        return link

    def forward(self, link, inputs, device):
        x, = inputs
        return link(x),

    def get_initializers(self):
        initialW = [initializers.Constant(1), 2]
        initial_bias = [initializers.Constant(2), 3,
            chainer.testing.link.InitializerArgument(None, 0)]
        return initialW, initial_bias
```

See also:

LinkTestCase *FunctionTestCase*

Methods

`__call__`(**args*, ***kwds*)
Call self as a function.

`addCleanup`(*function*, **args*, ***kwargs*)
Add a function, with arguments, to be called when the test is completed. Functions added are called on a

LIFO basis and are called after tearDown on test failure or success.

Cleanup items are called even if setUp fails (unlike tearDown).

addTypeEqualityFunc (*typeobj*, *function*)
Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Parameters

- **typeobj** – The data type to call this function on when both values are of the same type in assertEqual().
- **function** – The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

assertAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertAlmostEquals (***kwargs*)

assertCountEqual (*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset (*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertDictEqual (*d1*, *d2*, *msg=None*)

assertEqual (*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEquals (***kwargs*)

assertFalse (*expr*, *msg=None*)
Check that the expression is false.

assertGreater (*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance (*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone (*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone (*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess (*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual (*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs (*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual (*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

Objects that are equal automatically fail.

assertNotAlmostEquals (***kwargs*)

assertNotEqual (*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotEquals (***kwargs*)

assertNotIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance (*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex (*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNotRegexpMatches (***kwargs*)

assertRaises (*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex (*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRaisesRegexp (***kwargs*)

assertRegex (*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertRegexpMatches (***kwargs*)

assertSequenceEqual (*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)

A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)

Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)

A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)

Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

assert_(***kwargs*)

before_test(*test_name*)

check_forward_outputs(*outputs*, *expected_outputs*)

countTestCases()

create_link(*initializers*)

debug()
Run the test without collecting errors in a TestResult

defaultTestResult()

doCleanups()
Execute all cleanup functions. Normally called for you after tearDown.

fail(*msg=None*)
Fail immediately, with the given message.

failIf(***kwargs*)

failIfAlmostEqual(***kwargs*)

failIfEqual(***kwargs*)

failUnless(***kwargs*)

failUnlessAlmostEqual(***kwargs*)

failUnlessEqual(***kwargs*)

failUnlessRaises(***kwargs*)

forward(*link*, *inputs*, *device*)

generate_inputs()

generate_params()

get_initializers()

id()

run(*result=None*)

setUp()
Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

subTest(*msg=<object object>*, ***params*)
Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

tearDown()
Hook method for deconstructing the test fixture after testing it.

classmethod tearDownClass()
Hook method for deconstructing the class fixture after running all tests in the class.

test_initializers(*backend_config*)
Tests that the parameters of a links are correctly initialized.

__eq__(*other*)
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

backend_config = None

check_initializers_options = None

contiguous = None

longMessage = True

maxDiff = 640

param_names = ()

chainer.testing.LinkTestCase

class chainer.testing.**LinkTestCase**(**args*, ***kwargs*)
A base class for link forward and backward test cases.

Link test cases can inherit from this class to define a set of link tests for forward and backward computations.

Required methods

Each concrete class must at least override the following methods.

generate_params(self) Returns a tuple of initializers-likes. The tuple should contain an initializer-like for each initializer-like argument, i.e. the parameters to the link constructor. These will be passed to `create_link`.

create_link(self, initializers) Returns a link. The link should be initialized with the given initializer-likes `initializers`. `initializers` is a tuple of same length as the number of parameters.

generate_inputs(self) Returns a tuple of input arrays of type `numpy.ndarray`.

forward(self, link, inputs, device) Implements the target forward function. `link` is a link created by `create_link` and `inputs` is a tuple of *Variable*s. This method is expected to return the output *Variable*s with the same array types as the inputs. `device` is the device corresponding to the input arrays. A default implementation is provided for links that only takes the inputs defined in `generate_inputs` (wrapped in *Variable*s) and returns nothing but output *Variable*s in its forward computation.

Optional methods

Each concrete class may override the following methods depending on the skip flags `skip_forward_test` and `skip_backward_test`.

before_test(self, test_name) A callback method called before each test. Typically a skip logic is implemented by conditionally raising `unittest.SkipTest`. `test_name` is one of `'test_forward'` and `'test_backward'`.

forward_expected(self, link, inputs) Implements the expectation of the target forward function. `link` is the initialized link that was used to compute the actual forward which the results of this method will be compared against. The link is guaranteed to reside on the CPU. `inputs` is a tuple of `numpy.ndarray`s. This method is expected to return the output `numpy.ndarray`s. This method must be implemented if either `skip_forward_test` or `skip_backward_test` is `False` in which case forward or backward tests are executed.

generate_grad_outputs(self, outputs_template) Returns a tuple of output gradient arrays of type `numpy.ndarray`. `outputs_template` is a tuple of template arrays. The returned arrays are expected to have the same shapes and dtypes as the template arrays.

check_forward_outputs(self, outputs, expected_outputs) Implements check logic of forward outputs. Typically additional check can be done after calling `super().check_forward_outputs`. `outputs` and `expected_outputs` are tuples of arrays. In case the check fails, `LinkTestError` should be raised.

Attributes

The concrete class can override the following attributes to control the behavior of the tests.

param_names (tuple of str): A tuple of strings with all the names of the parameters that should be tested. E.g. `('gamma', 'beta')` for the batch normalization link. `()` by default.

skip_forward_test (bool): Whether to skip forward computation test. `False` by default.

skip_backward_test (bool): Whether to skip backward computation test. `False` by default.

dodge_nondifferentiable (bool): Enable non-differentiable point detection in numerical gradient calculation. If the data returned by `generate_params`, `create_link` and `generate_inputs` turns out to be a non-differentiable point, the test will repeatedly resample those until a differentiable point will be finally sampled. `False` by default.

numerical_grad_dtype (dtype): Input arrays are casted to this dtype when calculating the numerical gradients. It is `float64` by default, no matter what the original input dtypes were, to maximize precision.

contiguous (None or 'C'): Specifies the contiguousness of incoming arrays (i.e. inputs, parameters and gradients. If `None`, the arrays will be non-contiguous as long as possible. If `'C'`, the arrays will be C-contiguous. `None` by default.

Note: This class assumes `chainer.testing.inject_backend_tests()` is used together. See the example below.

Note: When implementing `LinkTestCase` and `LinkInitializersTestCase` to test both forward/backward and initializers, it is often convenient to refactor out common logic in a separate class.

Example

```
@chainer.testing.inject_backend_tests(
    None,
    [
        {},  # CPU
        {'use_cuda': True},  # GPU
    ])
class TestLinear(chainer.testing.LinkTestCase):

    param_names = ('W', 'b')

    def generate_params(self):
        initialW = numpy.random.uniform(
            -1, 1, (3, 2)).astype(numpy.float32)
        initial_bias = numpy.random.uniform(
            -1, 1, (3,)).astype(numpy.float32)
        return initialW, initial_bias

    def generate_inputs(self):
        x = numpy.random.uniform(
            -1, 1, (1, 2)).astype(numpy.float32)
        return x,

    def create_link(self, initializers):
        initialW, initial_bias = initializers
        link = chainer.links.Linear(
            2, 3, initialW=initialW, initial_bias=initial_bias)
        return link

    def forward(self, link, inputs, device):
        x, = inputs
        return link(x),
```

(continues on next page)

(continued from previous page)

```
    def forward_expected(self, link, inputs):
        W = link.W.array
        b = link.b.array
        x, = inputs
        expected = x.dot(W.T) + b
        return expected,
```

See also:

LinkInitializersTestCase *FunctionTestCase*

Methods

`__call__`(**args*, ***kwds*)
Call self as a function.

`addCleanup`(*function*, **args*, ***kwargs*)
Add a function, with arguments, to be called when the test is completed. Functions added are called on a LIFO basis and are called after tearDown on test failure or success.

Cleanup items are called even if setUp fails (unlike tearDown).

`addTypeEqualityFunc`(*typeobj*, *function*)
Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Parameters

- **`typeobj`** – The data type to call this function on when both values are of the same type in assertEqual().
- **`function`** – The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

`assertAlmostEqual`(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

If the two objects compare equal then they will automatically compare almost equal.

`assertAlmostEquals`(***kwargs*)

`assertCountEqual`(*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.

- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset (*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertDictEqual (*d1*, *d2*, *msg=None*)

assertEqual (*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEquals (***kwargs*)

assertFalse (*expr*, *msg=None*)
Check that the expression is false.

assertGreater (*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance (*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone (*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone (*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess (*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual (*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs (*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual (*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

Objects that are equal automatically fail.

assertNotAlmostEquals (***kwargs*)

assertNotEqual (*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotEquals (***kwargs*)

assertNotIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance (*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex (*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNotRegexpMatches (***kwargs*)

assertRaises (*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex (*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRaisesRegexp(***kwargs*)

assertRegex(*text, expected_regex, msg=None*)
Fail the test unless the text matches the regular expression.

assertRegexpMatches(***kwargs*)

assertSequenceEqual(*seq1, seq2, msg=None, seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1, set2, msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr, msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1, tuple2, msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning, *args, **kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional

and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

assert_(***kwargs*)

before_test(*test_name*)

check_forward_outputs(*outputs*, *expected_outputs*)

countTestCases()

create_link(*initializers*)

debug()
Run the test without collecting errors in a TestResult

defaultTestResult()

doCleanups()
Execute all cleanup functions. Normally called for you after tearDown.

fail(*msg=None*)
Fail immediately, with the given message.

failIf(***kwargs*)

failIfAlmostEqual(***kwargs*)

failIfEqual(***kwargs*)

failUnless(***kwargs*)

failUnlessAlmostEqual(***kwargs*)

failUnlessEqual(***kwargs*)

failUnlessRaises(***kwargs*)

forward(*link*, *inputs*, *device*)

forward_expected(*link*, *inputs*)

generate_grad_outputs(*outputs_template*)

generate_inputs()

generate_params()

id()

run(*result=None*)

setUp()
Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

subTest(*msg=<object object>*, ***params*)
Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

tearDown()
Hook method for deconstructing the test fixture after testing it.

classmethod tearDownClass()
Hook method for deconstructing the class fixture after running all tests in the class.

test_backward(*backend_config*)
Tests backward computation.

test_forward(*backend_config*)
Tests forward computation.

__eq__(*other*)
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Attributes

backend_config = None

check_backward_options = None

check_forward_options = None

contiguous = None

dodge_nondifferentiable = False

longMessage = True

maxDiff = 640

param_names = ()

skip_backward_test = False

skip_forward_test = False

4.20.6 Serialization testing utilities

Utilities for testing serializable objects.

chainer.testing.save_and_load	Saves `src` and loads it to `dst` using a de/serializer.
chainer.testing.save_and_load_hdf5	Saves `src` to an HDF5 file and loads it to `dst`.
chainer.testing.save_and_load_npz	Saves `src` to an NPZ file and loads it to `dst`.

chainer.testing.save_and_load

chainer.testing.**save_and_load**(*src*, *dst*, *filename*, *saver*, *loader*)
Saves `src` and loads it to `dst` using a de/serializer.

This function simply runs a serialization and deserialization to check if the serialization code is correctly implemented. The save and load are done within a temporary directory.

Parameters

- **src** – An object to save from.
- **dst** – An object to load into.
- **filename** (*str*) – File name used during the save/load.
- **saver** (*callable*) – Function that saves the source object.
- **loader** (*callable*) – Function that loads the file into the destination object.

chainer.testing.save_and_load_hdf5

`chainer.testing.save_and_load_hdf5(src, dst)`

Saves `src` to an HDF5 file and loads it to `dst`.

This is a short cut of `save_and_load()` using HDF5 de/serializers.

Parameters

- **src** – An object to save.
- **dst** – An object to load to.

chainer.testing.save_and_load_npz

`chainer.testing.save_and_load_npz(src, dst)`

Saves `src` to an NPZ file and loads it to `dst`.

This is a short cut of `save_and_load()` using NPZ de/serializers.

Parameters

- **src** – An object to save.
- **dst** – An object to load to.

4.20.7 Trainer Extension Testing Utilities

Utilities for testing *trainer extensions*.

`chainer.testing.get_trainer_with_mock_updater`	Returns a `Trainer` object with mock updater.

chainer.testing.get_trainer_with_mock_updater

`chainer.testing.get_trainer_with_mock_updater(stop_trigger=(10, 'iteration'), iter_per_epoch=10, extensions=None)`

Returns a `Trainer` object with mock updater.

The returned trainer can be used for testing the trainer itself and the extensions. A mock object is used as its updater. The update function set to the mock correctly increments the iteration counts (`updater.iteration`), and thus you can write a test relying on it.

Parameters

- **stop_trigger** – Stop trigger of the trainer.
- **iter_per_epoch** – The number of iterations per epoch.
- **extensions** – Extensions registered to the trainer.

Returns Trainer object with a mock updater.

4.20.8 Repeat decorators

These decorators have a decorated test run multiple times in a single invocation. Criteria of passing / failing of the test changes according to the type of decorators. See the documentation of each decorator for details.

chainer.testing.condition.repeat_with_success_at_least
chainer.testing.condition.repeat
chainer.testing.condition.retry

4.20.9 Unit test annotation

Decorators for annotating unit tests.

chainer.testing.attr.gpu	Decorator to indicate that GPU is required to run the test.
chainer.testing.attr.multi_gpu	Decorator to indicate number of GPUs required to run the test.
chainer.testing.with_requires	Run a test case only when given requirements are satisfied.
chainer.testing.fix_random	Decorator that fixes random numbers in a test.

chainer.testing.attr.gpu

chainer.testing.attr.**gpu**(*f*)

Decorator to indicate that GPU is required to run the test.

Tests can be annotated with this decorator (e.g., `@gpu`) to declare that one GPU is required to run.

chainer.testing.attr.multi_gpu

chainer.testing.attr.**multi_gpu**(*gpu_num*)

Decorator to indicate number of GPUs required to run the test.

Tests can be annotated with this decorator (e.g., `@multi_gpu(2)`) to declare number of GPUs required to run. When running tests, if `CHAINER_TEST_GPU_LIMIT` environment variable is set to value greater than or equals to 0, test cases that require GPUs more than the limit will be skipped.

chainer.testing.with_requires

chainer.testing.**with_requires**(**requirements*)

Run a test case only when given requirements are satisfied.

Example

This test case runs only when *numpy>=1.10* is installed.

```
>>> import unittest
>>> from chainer import testing
>>> class Test(unittest.TestCase):
...     @testing.with_requires('numpy>=1.10')
...     def test_for_numpy_1_10(self):
...         pass
```

Parameters **requirements** – A list of string representing requirement condition to run a given test case.

chainer.testing.fix_random

chainer.testing.**fix_random**()

Decorator that fixes random numbers in a test.

This decorator can be applied to either a test case class or a test method. It should not be applied within `condition.retry` or `condition.repeat`.

4.20.10 Parameterized test

Decorators for making a unit test parameterized.

chainer.testing.parameterize
chainer.testing.product
chainer.testing.product_dict
chainer.testing.inject_backend_tests

chainer.testing.parameterize

chainer.testing.**parameterize**(**params*)

chainer.testing.product

chainer.testing.**product**(*parameter*)

chainer.testing.product_dict

chainer.testing.**product_dict**(**parameters*)

chainer.testing.inject_backend_tests

chainer.testing.**inject_backend_tests**(*method_names*, *params*)

CHAPTER

FIVE

INSTALLATION

5.1 Recommended Environments

We recommend the following Linux distributions.

- Ubuntu 14.04 / 16.04 LTS (64-bit)
- CentOS 7 (64-bit)

Note: We are automatically testing Chainer on all the recommended environments above. We cannot guarantee that Chainer works on other environments including Windows and macOS (especially with CUDA support), even if Chainer may seem to be running correctly.

5.2 Requirements

You need to have the following components to use Chainer.

- **Python**
  - Supported Versions: 3.5.1+, 3.6.0+ and 3.7.0+.
- **NumPy**
  - Supported Versions: 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16 and 1.17.
  - NumPy will be installed automatically during the installation of Chainer.

Before installing Chainer, we recommend that you upgrade `setuptools` and `pip`:

```
$ pip install -U setuptools pip
```

Note: Python 2 is not supported in Chainer v7.x releases. Please consider migrating Python 3 or use Chainer v6.x, which is the last version that supports Python 2.

5.2.1 Hardware Acceleration Support

You can accelerate performance of Chainer by installing the following optional components.

- **NVIDIA CUDA / cuDNN**

- CuPy 5.0+
- See CuPy Installation Guide for instructions.

- **Intel CPU (experimental)**
  - iDeep 2.0.0.post3+
  - See *Tips and FAQs* for instructions.

5.2.2 Optional Features

The following packages are optional dependencies. Chainer can be installed without them, in which case the corresponding features are not available.

- **Image dataset support**
  - pillow 2.3+
  - Run `pip install pillow` to install.
- **HDF5 serialization support**
  - h5py 2.5+
  - Run `pip install h5py` to install.
- **Distributed Deep Learning using ChainerMN**
  - CUDA-aware MPI
  - mpi4py
  - See *ChainerMN installation guide* for installation instructions.

5.3 Install Chainer

5.3.1 Using pip

We recommend to install Chainer via pip:

```
$ pip install chainer
```

Note: Any optional dependencies (including CuPy) can be added after installing Chainer. Chainer automatically detects the available packages and enables/disables the optional features appropriately.

5.3.2 Using Tarball

The tarball of the source tree is available via `pip download chainer` or from the release notes page. You can install Chainer from the tarball:

```
$ pip install chainer-x.x.x.tar.gz
```

You can also install the development version of Chainer from a cloned Git repository:

```
$ git clone https://github.com/chainer/chainer.git
$ cd chainer
$ pip install .
```

5.3.3 Enable CUDA/cuDNN support

In order to enable CUDA support, you have to install CuPy manually. If you also want to use cuDNN, you have to install CuPy with cuDNN support. See CuPy's installation guide to install CuPy. Once CuPy is correctly set up, Chainer will automatically enable CUDA support.

You can refer to the following flags to confirm if CUDA/cuDNN support is actually available.

chainer.backends.cuda.available `True` if Chainer successfully imports `cupy`.

chainer.backends.cuda.cudnn_enabled `True` if cuDNN support is available.

5.3.4 Google Colaboratory

You can install Chainer and CuPy using the following snippet on Google Colaboratory:

```
!curl https://colab.chainer.org/install | sh -
```

See chainer/google-colaboratory for more details and examples.

5.4 Uninstall Chainer

Use pip to uninstall Chainer:

```
$ pip uninstall chainer
```

Note: When you upgrade Chainer, `pip` sometimes install the new version without removing the old one in `site-packages`. In this case, `pip uninstall` only removes the latest one. To ensure that Chainer is completely removed, run the above command repeatedly until `pip` returns an error.

5.5 Upgrade Chainer

Just use `pip` with `-U` option:

```
$ pip install -U chainer
```

5.6 Reinstall Chainer

If you want to reinstall Chainer, please uninstall Chainer and then install it. We recommend to use `--no-cache-dir` option as `pip` sometimes uses cache:

```
$ pip uninstall chainer
$ pip install chainer --no-cache-dir
```

5.7 Run Chainer with Docker

We are providing the official Docker image. Use nvidia-docker command to run Chainer image with GPU. You can login to the environment with bash, and run the Python interpreter:

```
$ nvidia-docker run -it chainer/chainer /bin/bash
```

Or run the interpreter directly:

```
$ nvidia-docker run -it chainer/chainer /usr/bin/python
```

5.8 FAQ

5.8.1 Warning message "cuDNN is not enabled" appears

You failed to build CuPy with cuDNN. If you don't need cuDNN, ignore this message. Otherwise, retry to install CuPy with cuDNN. `pip install -vvvv` option helps you. There is no need of re-installing Chainer itself. See CuPy's installation guide for more details.

5.8.2 CuPy always raises `cupy.cuda.compiler.CompileException`

See FAQ section of CuPy's installation guide for details.

5.8.3 h5py installation failed

If the installation failed with error saying `hdf5.h is not found`, you need to install `libhdf5` first. The way to install it depends on your environment:

```
# Ubuntu 14.04/16.04
$ apt-get install libhdf5-dev

# CentOS 7
$ yum -y install epel-release
$ yum install hdf5-devel
```

Note that `h5py` is not required unless you need HDF5 serialization support.

CHAPTER

SIX

CHAINERX DOCUMENTATION

Warning: This feature is still in the earliest stage of its development. The behavior and interface are subject to change.

ChainerX is an ndarray implementation with Define-by-Run automatic differentiation capability. It roughly corresponds to "NumPy/CuPy + Chainer Variable", while some additional features follow:

- **Speed**: The whole ndarray and autograd implementation is written in C++, with a thin Python binding. It lowers the overhead existing in the pure Python implementation of Chainer.
- **Extensibility**: The backend is pluggable so that it is much easier to add a support of new devices.

The speed is best achieved by directly using ChainerX APIs, while it also provides a compatibility layer through the conventional `chainer.Variable` interface for easier adoption of ChainerX in existing projects. See *ChainerX Tutorial* for more details.

6.1 Installation

ChainerX, or `chainerx`, can be installed as a top level Python package along with Chainer by configuring the environment variables below.

Note: Chainer must currently be installed from source in order to include ChainerX, but this is expected to change in the near future.

6.1.1 Installing from source

The following environment variables are available for building ChainerX from source.

Environment variable	Description
CHAINER_BUILD_CHAINERX	1 to build the chainerx package along with chainer. 0 to skip. Default is 0.
CHAINERX_BUILD_CUDA	1 to build chainerx with CUDA support. 0 to skip. Default is 0. See also *CUDA support* section below.
CHAINERX_ENABLE_BLAS	1 to enable BLAS, 0 to disable it. Default is 1. If BLAS is enabled, it is searched for and used if found. If not found, ChainerX will behave as if BLAS was disabled and use a basic implementation instead.
CHAINERX_ENABLE_LAPACK	1 to enable LAPACK, 0 to disable it. Default is 1. If LAPACK is enabled, it is searched for and used if found. If not found, ChainerX will behave as if LAPACK was disabled and may cause runtime errors.

Simply run `pip install --pre chainer` after configuring the above environment variables. See *Examples* below.

6.1.2 CUDA support

When installing with the CUDA support, you also need to specify the cuDNN installation path.

You can specify either of the following environment variables to specify where to look for cuDNN installation.

Environment variable	Description
CUDNN_ROOT_DIR	Path to your cuDNN installation.
CHAINERX_CUDNN_USE_CUPY	1 to search for cuDNN library and include files in existing CuPy installation. Only applicable for CuPy installed via wheel (binary) distribution. Other variables related to cuDNN paths (such as CUDNN_ROOT_DIR) are ignored. Be warned that the resulting executable will be invalidated if CuPy is uninstalled, moved or replaced.

To support the *NumPy/CuPy fallback* mechanism, currently ChainerX with the CUDA support requires CuPy to be installed together.

See also:

CuPy installation guide

6.1.3 Examples

Install ChainerX without CUDA support:

```
$ export CHAINER_BUILD_CHAINERX=1
$ export MAKEFLAGS=-j8  # Using 8 parallel jobs.
$ pip install --pre chainer
```

Install ChainerX depending on CuPy wheel distribution:

```
$ pip install --pre cupy_cuda101  # Note: Choose the proper CUDA SDK version number.
$ export CHAINER_BUILD_CHAINERX=1
$ export CHAINERX_BUILD_CUDA=1
$ export CHAINERX_CUDNN_USE_CUPY=1
$ export MAKEFLAGS=-j8  # Using 8 parallel jobs.
$ pip install --pre chainer
```

Install ChainerX with CuPy built from source:

```
$ export CHAINER_BUILD_CHAINERX=1
$ export CHAINERX_BUILD_CUDA=1
$ export CUDNN_ROOT_DIR=path/to/cudnn
$ export MAKEFLAGS=-j8  # Using 8 parallel jobs.
$ pip install --pre cupy
$ pip install --pre chainer
```

6.2 ChainerX Tutorial

ChainerX, or *chainerx*, is meant to be a drop-in replacement for NumPy and CuPy, with additional operations specific to neural networks. As its core is implemented in C++, you can reduce the Python overhead for both the forward and backward passes compared to Chainer, speeding up your training and inference. This section will guide you through the essential APIs of Chainer to utilize ChainerX, but also how to use ChainerX on its own.

6.2.1 Introduction to ChainerX

The module *chainerx* aims to support a NumPy compatible interface with additional operations specific to neural networks. It for instance provides `chainerx.conv()` for N-dimensional convolutions and `chainerx.batch_norm()` for batch normalization. Additionally, and most importantly, the array in ChainerX *chainerx.ndarray*, distinguishes itself from NumPy and CuPy arrays in the following two aspects.

Automatic differentiation Graph construction and backpropagation is built into the array, meaning that any function, including the NumPy-like functions, can be backpropagated through. In Chainer terms, it is a NumPy/CuPy array with *chainer.Variable* properties.

Device agnostic Arrays can be allocated on any device belonging to any backend, in contrast to NumPy/CuPy arrays which are implemented for specific computing platforms (i.e. CPUs/GPUs respectively).

These differences are explained more in details by the sections further down.

The array `chainerx.ndarray`

The following example demonstrates how you can create an array and access its most basic attributes. Note that the APIs are identical to that of NumPy and CuPy. Other array creation routines including `chainerx.ones()`, `chainerx.ones_like()` and `chainerx.random.normal()` are all listed in *here*.

```
import chainerx as chx

x = chx.array([[0, 1, 2], [3, 4, 5]], dtype=chx.float32)

x.shape  # (2, 3)
x.dtype  # dtype('float32')
x.size  # 6
x.ndim  # 2
```

Backends and devices

Chainer distinguishes between CPU and GPU arrays using NumPy and CuPy but ChainerX arrays may be allocated on any device on any backend. You can specify the device during instantiation or transfer the array to a different device after it has been created.

```
x = chx.array([1, 2, 3])
x.device  # native:0

x = chx.array([1, 2, 3], device='cuda:0')
x.device  # cuda:0

x = x.to_device('cuda:1')
x.device  # cuda:1
```

The left-hand-side of the colon shows the name of the backend to which the device belongs. `native` in this case refers to the CPU and `cuda` to CUDA GPUs. The integer on the right-hand-side shows the device index. Together, they uniquely identify a physical device on which an array is allocated.

If you do not want to specify the device each time you create an array, it is possible to change the default device with `chainerx.using_device()`.

```
with chx.using_device('cuda:0')
    x = chx.array([1, 2, 3])
x.device  # cuda:0
```

Note: Currently, two backends are built into ChainerX.

1. The `native` backend, which is built by default.
2. The `cuda` backend which is optional (See *installation*).

This backend abstraction allows developers to implement their own backends and plug them into ChainerX to perform computations on basically any other platform.

Array operations and backpropagation

Arrays support basic arithmetics and can be passed to functions just as you would expect. By marking an array to require gradients with `chainerx.ndarray.require_grad()`, further computations involving that array will construct a computational graph allowing backpropagation directly from the array. The following code shows how you could implement an affine transformation and backpropgate through it to compute the gradient of the output w.r.t. the input weight and bias.

```
x = chx.ones(784, dtype=chx.float32)
W = chx.random.normal(size=(784, 1000)).astype(chx.float32).require_grad()
b = chx.random.normal(size=(1000)).astype(chx.float32).require_grad()

y = x.dot(W) + b

y.grad = chx.ones_like(y)  # Initial upstream gradients, i.e. `grad_outputs`.
y.backward()

assert type(W.grad) is chx.ndarray
assert type(b.grad) is chx.ndarray
```

Note: The code above is device agnostic, meaning that you can execute it on any backend by simply wrapping the code with a `chainerx.using_device()`.

6.2.2 Relation to Chainer

A *chainerx.ndarray* can be wrapped in a *chainer.Variable* and passed to any existing Chainer code.

```
var = ch.Variable(x)  # x is a chainerx.ndarray.

# Your Chainer code...
```

When further applying functions to the `var`, the computational graph is recorded in the underlying ndarray in C++ implementation, not in the *chainer.Variable* or the *chainer.FunctionNode*, as in the conventional Chainer. This eliminates the heavy Python overhead of the graph construction. Similarly, calling *chainer.Variable.backward()* on any resulting variable will delegate the work to C++ by calling `chainerx.ndarray.backward()` spending no time in the Python world.

NumPy/CuPy fallback

As the features above require ChainerX to provide an implementation corresponding to every *chainer.FunctionNode* implementation in Chainer, ChainerX utilizes a fallback mechanism while gradually extending the support. This approach is taken because the integration with Chainer takes time and we do not want existing Chainer users to have to make severe changes to their code bases in order to try ChainerX. The fallback logic simply casts the *chainerx.ndarray*s inside the *chainer.Variable* to `numpy.ndarray`s or `cupy.ndarray`s (without copy) and calls the forward and backward methods respectively.

Run your Chainer code with ChainerX

In order to utilize *chainerx*, you first need to transfer your model to a ChainerX device using *chainer.Link.to_device()*. This is a new method that has been introduced to replace *chainer.Link.to_cpu()* and *chainer.Link.to_gpu()*, extending device transfer to arbitrary devices. Similarly, you have to transfer the data (*chainer.Variable*s) to the same device before feeding them to the model.

Will my FunctionNode work with ChainerX?

Our expectation is that it should work because of the fallback mechanism explained above, but in practice you may need some occasional fixes, depending on how the function was implemented. Also, you will not see any performance improvements from the fallback (but most likely a degradation because of the additional conversions).

To support ChainerX with your *chainer.FunctionNode*, you need to implement *chainer.FunctionNode.forward_chainerx()* with the same signature as *chainer.FunctionNode.forward()*, but where given inputs are of type *chainerx.ndarray*. It is expected to return a `tuple` just like *chainer.FunctionNode.forward()*.

The example below shows how *chainer.functions.matmul()* is extended to support ChainerX. Note that `chainer.Fallback` can be returned in case the function cannot be implemented using ChainerX functions. This is also the default behavior in case the method is not implemented at all.

```
class MatMul(function_node.FunctionNode):

    def forward_chainerx(self, x):
        a, b = x
        if self.transa or self.transb or self.transc:
            return chainer.Fallback
        if a.dtype != b.dtype:
            return chainer.Fallback
```

(continues on next page)

(continued from previous page)

```
        if a.ndim != 2 or b.ndim != 2:
            return chainer.Fallback
        if self.dtype is not None and self.dtype != a.dtype:
            return chainer.Fallback
        return chainerx.dot(a, b),  # Fast C++ implementation
```

6.3 Limitations

There are some non-obvious limitations in ChainerX:

- ChainerX only supports a limited set of dtypes: `bool_` `int8` `int16` `int32` `int64` `uint8` `float32` `float64`.
- Operations with mixed dtypes are not supported. You need to explicitly convert dtypes using either `chainerx.astype()` or *`F.cast()`*.
- True division of Python, where 2/3 returns .66 rather than 0, is not supported yet. Given an ndarray `a` of the dtype `int32`, `a / a` does not return an array of `float64`, but returns an array of `int32`.
- Only a limited set of Chainer *`function`*s are well tested with the ChainerX integration.
- ChainerX CUDA backend requires cuDNN. See *installation* for details.
- As ChainerX *`array`*s have a computational graph in their own, some operations are prohibited for safety:
  - Unless an array is free from the computational graph, in-place modification of its data is prohibited.

    ```
    a = chainerx.zeros((2,), chainerx.float32)
    a.require_grad()  # install the computational graph on `a`.

    a += 1  # ! error
    ```

    The reason of this limitation is that, as backward operations may depend on the value of `a`, the backward gradients might be unexpectedly affected if it would be altered.

    You may circumvent this limitation by making a disconnected view:

    ```
    # A memory-shared view of `a` which is disconnected from the computational␣
    ↪graph of `a`.
    b = a.as_grad_stopped()

    b += 1
    ```

    Note however that this operation is inherently dangerous. You should be super careful to ensure that that does not affect backward computations.

    Note also that we may restrict further in the future so that even in-place modification on a disconnected view is only allowed if it is actually safe.

  - If an array is wrapped with a *`Variable`* with `requires_grad=True` (which is default), you won't be able to re-assign the array:

    ```
    a = chainerx.zeros((2,), chainerx.float32)
    b = chainerx.zeros((2,), chainerx.float32)
    var = chainer.Variable(a)

    var.array = b  # ! error
    ```

You may circumvent this by using in-place assignment on `var.array`:

```
var.array[:] = b
```

This workaround may also be dangerous just as in the previous limitation.

6.4 Reference

6.4.1 Multi-Dimensional Array (ndarray)

chainerx.ndarray	Dummy class for type testing.

chainerx.ndarray

class chainerx.**ndarray**(**args*, ***kwargs*)
Dummy class for type testing.

Methods

__eq__()
Return self==value.

__ne__()
Return self!=value.

__lt__()
Return self<value.

__le__()
Return self<=value.

__gt__()
Return self>value.

__ge__()
Return self>=value.

Utility functions

chainerx.to_numpy

6.4.2 Array Operations

Array creation routines

chainerx.empty
chainerx.empty_like
chainerx.eye
chainerx.identity

Continued on next page

Table 3 – continued from previous page

chainerx.ones
chainerx.ones_like
chainerx.zeros
chainerx.zeros_like
chainerx.full
chainerx.full_like
chainerx.array
chainerx.asarray
chainerx.asanyarray
chainerx.ascontiguousarray
chainerx.copy
chainerx.frombuffer
chainerx.fromfile
chainerx.fromfunction
chainerx.fromiter
chainerx.fromstring
chainerx.loadtxt
chainerx.arange
chainerx.linspace
chainerx.diag
chainerx.diagflat
chainerx.tri
chainerx.tril
chainerx.triu

Activation functions

chainerx.log_softmax
chainerx.tanh
chainerx.relu
chainerx.sigmoid
chainerx.slstm
chainerx.tree_lstm

Array manipulation routines

chainerx.reshape
chainerx.ravel
chainerx.transpose
chainerx.broadcast_to
chainerx.squeeze
chainerx.asarray
chainerx.ascontiguousarray
chainerx.concatenate
chainerx.stack
chainerx.hstack
chainerx.vstack
chainerx.dstack
chainerx.atleast_2d

Continued on next page

Table 5 – continued from previous page

chainerx.atleast_3d
chainerx.split
chainerx.dsplit
chainerx.vsplit
chainerx.swapaxes
chainerx.repeat
chainerx.expand_dims
chainerx.flip
chainerx.fliplr
chainerx.flipud
chainerx.moveaxis

Evaluation routines

chainerx.accuracy

Indexing routines

chainerx.take
chainerx.where
chainerx.nonzero

Linear algebra

chainerx.dot
chainerx.linalg.cholesky
chainerx.linalg.qr
chainerx.linalg.svd
chainerx.linalg.solve
chainerx.linalg.inv
chainerx.linalg.pinv

Logic functions

chainerx.all
chainerx.any
chainerx.isinf
chainerx.isnan
chainerx.logical_and
chainerx.logical_or
chainerx.logical_xor
chainerx.logical_not
chainerx.greater
chainerx.greater_equal
chainerx.less
chainerx.less_equal
chainerx.equal

Continued on next page

Table 9 – continued from previous page

chainerx.not_equal

Loss functions

chainerx.absolute_error
chainerx.squared_error
chainerx.huber_loss
chainerx.gaussian_kl_divergence

Mathematical functions

chainerx.negative
chainerx.add
chainerx.subtract
chainerx.multiply
chainerx.divide
chainerx.mod
chainerx.remainder
chainerx.sum
chainerx.maximum
chainerx.minimum
chainerx.exp
chainerx.log
chainerx.log10
chainerx.log2
chainerx.log1p
chainerx.logsumexp
chainerx.log_softmax
chainerx.sqrt
chainerx.sin
chainerx.cos
chainerx.tan
chainerx.arcsin
chainerx.arccos
chainerx.arctan
chainerx.arctan2
chainerx.sinh
chainerx.cosh
chainerx.tanh
chainerx.arcsinh
chainerx.arccosh
chainerx.square
chainerx.clip
chainerx.fabs
chainerx.sign
chainerx.ceil
chainerx.floor
chainerx.bitwise_and
chainerx.bitwise_or

Continued on next page

Table 11 – continued from previous page

chainerx.bitwise_xor
chainerx.left_shift
chainerx.right_shift

Random sampling

chainerx.random.normal
chainerx.random.uniform

Sorting, searching, and counting

chainerx.argmax
chainerx.argmin

Statistics

chainerx.amax
chainerx.mean
chainerx.var

Connection

chainerx.conv
chainerx.conv_transpose
chainerx.linear
chainerx.lstm

Normalization

chainerx.batch_norm
chainerx.fixed_batch_norm

Pooling

chainerx.max_pool
chainerx.average_pool

RNN

chainerx.n_step_lstm
chainerx.n_step_bilstm
chainerx.n_step_gru
chainerx.n_step_bigru
chainerx.n_step_rnn

Continued on next page

Table 18 – continued from previous page

chainerx.n_step_birnn

6.4.3 Context

chainerx.Context

6.4.4 Backend and Device

ChainerX adds a level of abstraction between the higher level array operations and the lower level computations and resource management. This abstraction is managed by the `Backend` and the `Device` classes. Native (CPU) and CUDA backends are two concrete implementations currently provided by ChainerX but the abstraction allows you to plug any backend into the framework.

Backend

chainerx.Backend
chainerx.get_backend

Device

chainerx.Device
chainerx.get_device
chainerx.get_default_device
chainerx.set_default_device
chainerx.using_device

6.4.5 Utilities for Backpropagation

chainerx.backward
chainerx.no_backprop_mode
chainerx.force_backprop_mode
chainerx.is_backprop_required

6.5 Contribution Guide

This is a guide aimed towards contributors of ChainerX which is mostly implemented in C++. It describes how to build the project and how to run the test suite so that you can get started contributing.

Note: Please refer to the *Chainer Contribution Guide* for the more general contribution guideline that is not specific to ChainerX. E.g. how to download the source code, manage git branches, send pull requests or contribute to Chainer's Python code base.

Note: There is a public ChainerX Product Backlog.

6.5.1 Building the shared library

You can build the C++ ChainerX project to generate a shared library similar to any other cmake project. Run the following command from the root of the project to generate `chainerx_cc/build/chainerx/libchainerx.so`,

```
$ mkdir chainerx_cc/build
$ cd chainerx_cc/build
$ cmake ..
$ make
```

The CUDA support is enabled by, either setting `CHAINERX_BUILD_CUDA=1` as an environment variable or specifying `-DCHAINERX_BUILD_CUDA=1` in `cmake`. When building with the CUDA support, either the `CUDNN_ROOT_DIR` environment variable or `-DCUDNN_ROOT_DIR` is required to locate the cuDNN installation path.

Note: CUDA without cuDNN is currently not supported.

Then, to install the headers and the library, run:

```
$ make install
```

You can specify the installation path using the prefix `-DCMAKE_INSTALL_PREFIX=<...>` in `cmake`.

6.5.2 Running the test suite

The test suite can be built by passing `-DCHAINERX_BUILD_TEST=ON` to `cmake`. It is not built by default. Once built, run the suite with the following command from within the `build` directory.

```
$ cd chainerx_cc/build
$ ctest -V
```

6.5.3 Coding standards

The ChainerX C++ coding standard is mostly based on the Google C++ Style Guide and principles.

Formatting

ChainerX is formatted using clang-format. To fix the formatting in-place, run the following command from `chainerx_cc` directory:

```
$ cd chainerx_cc
$ scripts/run-clang-format.sh --in-place
```

Lint checking

ChainerX uses the cpplint and clang-tidy for lint checking. Note that clang-tidy requires that you've finished running `cmake`. To run cpplint, run `scripts/run-cpplint.sh` from `chainerx_cc` directory:

```
$ cd chainerx_cc
$ scripts/run-cpplint.sh
```

To run clang-tidy, run `make clang-tidy` from the build directory:

```
$ cd chainerx_cc/build
$ make clang-tidy
```

6.5.4 Thread sanitizer

The thread sanitizer can be used to detect thread-related bugs, such as data races. To enable the thread sanitizer, pass `-DCHAINERX_ENABLE_THREAD_SANITIZER=ON` to `cmake`.

You can run the test with `ctest -V` as usual and you will get warnings if the thread sanitizer detects any issues.

CUDA runtime is known to cause a thread leak error as a false alarm. In such case, disable the thread leak detection using environment variable `TSAN_OPTIONS='report_thread_leaks=0'`.

6.5.5 Python contributions and unit tests

To test the Python binding, run the following command at the repository root:

```
$ pytest
```

The above command runs all the tests in the repository, including Chainer and ChainerMN. To run only ChainerX tests, specify the test directory:

```
$ pytest tests/chainerx_tests
```

Run tests with coverage:

```
$ pytest --cov --no-cov-on-fail --cov-fail-under=80 tests/chainerx_tests
```

Run tests without CUDA GPU:

```
$ pytest -m 'not cuda' tests/chainerx_tests
```

6.6 Tips and FAQs

6.6.1 Can I use ChainerX without Chainer?

Yes, it is possible. See the code samples below.

- Train an MLP with MNIST dataset (chainerx_cc/examples/mnist_py)
- Train a CNN with ImageNet dataset (chainerx_cc/examples/imagenet_py)

6.6.2 What does the C++ interface look like?

It is almost identical to the Python interface with a 1-to-1 mapping. The interface is still subject to change, but there is an example code:

- Train an MLP with MNIST dataset in C++ (chainerx_cc/examples/mnist)

6.6.3 GPU memory consumption is too high when used with CuPy

Both ChainerX and CuPy use their own GPU memory pools, meaning that GPU memory is not efficiently utilized (unused memory is kept without being freed by both ChainerX and CuPy). You can run your script after setting the environment variable `CHAINERX_CUDA_CUPY_SHARE_ALLOCATOR` to `1` to use the experimental feature which makes sure that both ChainerX and CuPy share the same memory pool, hence reducing your peak GPU memory-usage. You may also invoke `chainerx._cuda.cupy_share_allocator` instead of setting the environment variable for the same effect. In this case, it is recommended that you call the function prior to any GPU memory allocation.

CHAPTER

SEVEN

DISTRIBUTED DEEP LEARNING WITH CHAINERMN

ChainerMN enables multi-node distributed deep learning with the following features:

- **Scalable** — it makes full use of the latest technologies such as NVIDIA NCCL and CUDA-Aware MPI,
- **Flexible** — even dynamic neural networks can be trained in parallel thanks to Chainer's flexibility, and
- **Easy** — minimal changes to existing user code are required.

This blog post provides our benchmark results using up to 128 GPUs.

ChainerMN can be used for both inner-node (i.e., multiple GPUs inside a node) and inter-node settings. For inter-node settings, we highly recommend to use high-speed interconnects such as InfiniBand.

ChainerMN examples are available on GitHub. These examples are based on the examples of Chainer and the differences are highlighted.

7.1 Installation

7.1.1 Installation Guide

Requirements

ChainerMN depends on the following software libraries: CUDA-Aware MPI, NVIDIA NCCL, and a few Python packages including CuPy and MPI4py.

Note: In Chainer v5, ChainerMN became a part of Chainer package. Installing Chainer (`pip install chainer`) automatically makes ChainerMN available. Note that you still need to separately install requirements described below to actually run code using ChainerMN.

Before upgrading from Chainer v4 to v5 or later, make sure to remove existing `chainermn` package (`pip uninstall chainermn`).

CUDA-Aware MPI

ChainerMN relies on MPI. In particular, for efficient communication between GPUs, it uses CUDA-aware MPI. For details about CUDA-aware MPI, see this introduction article. (If you use only the CPU mode, MPI does not need to be CUDA-Aware. See *Installation on Non-GPU Environments* for more details.)

The CUDA-aware features depend on several MPI packages, which need to be configured and built properly. The following are examples of Open MPI and MVAPICH.

Open MPI (for details, see Open MPI's official instructions):

```
$ ./configure --with-cuda
$ make -j4
$ sudo make install
```

MVAPICH (for details, see Mvapich's official instructions):

```
$ ./configure --enable-cuda
$ make -j4
$ sudo make install
$ export MV2_USE_CUDA=1  # Should be set all the time when using ChainerMN
```

NCCL

Note: If you are installing CuPy using wheels (i.e., `pip install cupy-cudaXX` where `XX` is the CUDA version), you don't have to install NCCL manually. The latest NCCL 2.x library is bundled with CuPy wheels.

See CuPy Installation Guide for the detailed steps to install CuPy.

To enable efficient intra- and inter-node GPU-to-GPU communication, we use NVIDIA Collective Communications Library (NCCL). See NCCL's official instructions for installation.

ChainerMN requires NCCL even if you have only one GPU per node. The only exception is when you run ChainerMN on CPU-only environments. See *Installation on Non-GPU Environments* for more details.

Note: We reccomend NCCL 2 but NCCL 1 can be used. However, for NCCL 1, `PureNcclCommunicator` is not supported in ChainerMN. If you use NCCL 1, please properly configure environment variables to expose NCCL both when you install and use ChainerMN. Typical configurations should look like the following:

```
export NCCL_ROOT=<path to NCCL directory>
export CPATH=$NCCL_ROOT/include:$CPATH
export LD_LIBRARY_PATH=$NCCL_ROOT/lib/:$LD_LIBRARY_PATH
export LIBRARY_PATH=$NCCL_ROOT/lib/:$LIBRARY_PATH
```

If you change the version of NCCL installed, you have to reinstall CuPy. Because, current ChainerMN applies CuPy to use NCCL. See CuPy official instructions for reinstalltion.

MPI4py

You can install MPI4py by:

```
$ pip install mpi4py
```

Please make be sure to properly configure environment variables so that MPI is available at installation time, because MPI4py links to MPI library at installation time. In particular, if you have multiple MPI implementations installed in your environment, please expose the implementation that you want to use both when you install and use ChainerMN.

As of writing, MPI4py does not support Open MPI 4.x. Please use versions from the *Tested Environments* section below.

CuPy

Chainer and ChainerMN rely on CuPy to use GPUs. Please refer to CuPy Installation Guide for the detailed steps to install CuPy.

In most cases it is recommended that you install CuPy using wheel distribution (precompiled binary) rather than source distribution. If you are installing from source, NCCL library must be installed before installing CuPy to enable NCCL feature in CuPy. Refer to *NCCL* for the installation steps of NCCL library. See *Check if NCCL is enabled in CuPy*, if you want to check whether NCCL is enabled in your CuPy.

Chainer and ChainerMN can be installed without CuPy, in which case the corresponding features are not available. See *Installation on Non-GPU Environments* for more details.

Tested Environments

We tested ChainerMN on all the following environments.

- OS
  - Ubuntu 14.04 LTS 64bit
  - Ubuntu 16.04 LTS 64bit
- Python 2.7.13, 3.5.1, 3.6.1
- MPI
  - Open MPI 2.1.6, 3.0.4, 3.1.4
- MPI4py 3.0.0
- NCCL 2.3.2 2.4.2

Note: Note that the following versions of Open MPI have some bugs that might cause ChainerMN programs to hang: 3.0.[0-2] and 3.1.[0-2]. For more details, see Open MPI Issue #3972 and Chainer Issue #5740 .

Also, mpi4py does not support Open MPI 4.0.x.

Installation on Non-GPU Environments

Users who want to try ChainerMN in CPU-only environment may skip installation of CuPy. Non-GPU set up may not be performant as GPU-enabled set up, but would be useful for testing or debugging training program in non-GPU environment such as laptops or CI jobs.

In this case, the MPI does not have to be CUDA-aware. Only `naive` communicator works with the CPU mode.

7.1.2 Step-by-Step Troubleshooting

This section is a step-by-step troubleshooting guide for ChainerMN. Please follow these steps to identify and fix your problem.

We assume that you are using Linux or another Unix-like environment.

Single-node environment

Basic MPI installation

Although ChainerMN stands for "Chainer MultiNode," it is good to start from single-node execution. First of all, you need MPI. If MPI is correctly installed, you will see the **mpicc** and **mpiexec** commands in your PATH.

Below is an example of the output from Mvapich on Linux.:

```
$ which mpicc
/usr/local/bin/mpicc

$ mpicc -show
gcc -I/usr/local/include ...(snip)... -lmpi

$ which mpiexec
/usr/local/bin/mpiexec

$ mpiexec --version
HYDRA build details:
Version:                                 3.1.4
Release Date:                            Wed Sep  7 14:33:43 EDT 2016
CC:                              gcc
CXX:                             g++
F77:
F90:
Configure options:  (snip)
Process Manager:                         pmi
Launchers available:                     ssh rsh fork slurm ll lsf sge manual persist
Topology libraries available:            hwloc
Resource management kernels available:   user slurm ll lsf sge pbs cobalt
Checkpointing libraries available:
Demux engines available:                 poll select
```

If you see any error in above commands, please go back to the *CUDA-Aware MPI* and check your MPI installation.

Check what MPI you are using

In *CUDA-Aware MPI*, we mention both of *Open MPI* and *Mvapich*. If the MPI is provided by the system administrator and you are not really sure which MPI you are using, check the output of *mpiexec –version*.

- If the output contains *HYDRA*, then it's MVAPICH (or possibly MPICH).
- If the output contains *OpenRTE*, then it's Open MPI.

However, in such a case, you should make sure that the MPI is *CUDA-aware*, as mentioned below. We recommend to build your own MPI.

Check if MPI is CUDA-aware

Your MPI must be configured as *CUDA-aware*. You can use the following C program to check it.

```
/* check_cuda_aware.c */
#include <assert.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
#include <mpi.h>
#include <cuda_runtime.h>

#define CUDA_CALL(expr) do {                    \
  cudaError_t err;                              \
  err = expr;                                   \
  assert(err == cudaSuccess);                   \
} while(0)

int main(int argc, char **argv) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int *sendbuf_d = NULL;
  int *recvbuf_d = NULL;

  CUDA_CALL(cudaMalloc((void**)&sendbuf_d, sizeof(int)));
  CUDA_CALL(cudaMalloc((void**)&recvbuf_d, sizeof(int)));
  CUDA_CALL(cudaMemcpy(sendbuf_d, &rank, sizeof(int), cudaMemcpyDefault));

  MPI_Reduce(sendbuf_d, recvbuf_d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank == 0) {
    int sum = -1;
    CUDA_CALL(cudaMemcpy(&sum, recvbuf_d, sizeof(int), cudaMemcpyDefault));
    if (sum == (size-1) * size / 2) {
      printf("OK.\n");
    } else {
      printf("Error.\n");
    }
  }

  cudaFree(sendbuf_d);
  cudaFree(recvbuf_d);

  MPI_Finalize();
}
```

Save the code to a file named `check_cuda_aware.c`. You can compile and run it with the following command.:

```
$ export MPICH_CC=nvcc   # if you use Mvapich
$ export OMPI_CC=nvcc    # if you use Open MPI
$ $(mpicc -show check_cuda_aware.c -arch sm_53 | sed -e 's/-Wl,/-Xlinker /g' | sed -e
↪'s/-pthread/-Xcompiler -pthread/')
$ ./a.out
OK.
```

If the proglam prints *OK.*, your MPI is correctly configured.

Check mpi4py

Next, let's check that mpi4py is correctly installed. You can use the following script to check it:

```
# coding: utf-8
import os
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

for i in range(size):
  if i == rank:
    print("{} {}".format(os.uname()[1], i))
  comm.Barrier()
```

Save the script into a file named `check_mpi4py.py` and run it. The output from the script should look like this.:

```
$ mpiexec -np 4 python check_mpi4py.py
host00 0
host00 1
host00 2
host00 3
```

The script prints hostnames and ranks (process id in MPI) from each MPI process in a sequential manner. *host00* is the host name of the machine your are running the process. If you get an output like below, it indicates something is wrong with your installation.:

```
# Wrong output !
$ mpiexec -n 4 python check_mpi4py.py
host00 0
host00 0
host00 0
host00 0
```

A common problem is that the **mpicc** used to build `mpi4py` and **mpiexec** used to run the script are from different MPI installations.

Finally, run **pytest** to check the single-node configuration is ready.:

```
$ git clone git@github.com:chainer/chainer.git
Cloning into 'chainer'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 168242 (delta 1), reused 2 (delta 0), pack-reused 168235
Receiving objects: 100% (168242/168242), 41.15 MiB | 1.65 MiB/s, done.
Resolving deltas: 100% (123696/123696), done.
Checking connectivity... done.
$ cd chainer/
$ pytest tests/chainermn_tests/
......S.S...S.S...S.S...S.S.........SS
----------------------------------------------------------------------
Ran 38 tests in 63.083s

OK (SKIP=10)
```

Check if NCCL is enabled in CuPy

CuPy requires NCCL to be enabled. You can check it with the following command.:

```
$ python -c 'from cupy.cuda import nccl'
```

If you get an output like below, NCCL is not enabled in CuPy. Please check the installation guide of CuPy.:

```
Traceback (most recent call last):

 File "<string>", line 1, in <module>

ImportError: cannot import name 'nccl'
```

Multi-node environment

Check SSH connection and environment variables

To use ChainerMN on multiple hosts, you need to connect to computing hosts, including the one you are currently logged into, via ssh without password authentication (and preferably without username).:

```
$ ssh host00 'hostname'
host00   # without hitting the password

$ ssh host01 'hostname'
host01   # without hitting the password

...
```

You may get a message like this:

```
The authenticity of host 'host01 (xxx.xxx.xxx.xxx)' can't be established.
ECDSA key fingerprint is SHA256:haGUMcCeC5A8lGh1lpjpwL5dF4xCglZArhhxxxxxxxxx.
Are you sure you want to continue connecting (yes/no)?
```

This message appears when you log in a host for the first time. Just type *yes* and the message won't appear again. You need to repeat this process on all computing hosts.

Also, you need to pay attention to the environment variables on remote hosts. The MPI runtime connects to the remote hosts in *non-interactive* mode, and environment variables may differ from your interactive login sessions.:

```
$ ssh host00 'env' | grep LD_LIBRARY_PATH
# Check the values and compare it to the local value.

$ ssh host01 'env' | grep LD_LIBRARY_PATH
# Check the values and compare it to the local value.

...
```

In particular, check the following variables, which are critical to executing MPI programs:

- `PATH`
- `LD_LIBRARY_PATH`
- `MV2_USE_CUDA` (if you use MVAPICH)
- `MV2_SMP_USE_CMA` (if you use MVAPICH)

Besides, you need to make sure the same **mpiexec** binary is used to run MPI programs.:

```
$ ssh host00 'which mpiexec'
/usr/local/bin/mpiexec

$ ssh host01 'which mpiexec'
/usr/local/bin/mpiexec
```

All the commands should give the same **mpiexec** binary path.

Program files and data

When you run MPI programs, all hosts must have the same Python binary and script files in the same path. First, check that the python binary and version are identical among hosts. Be careful if you are using *pyenv* or *Anaconda*.:

```
$ ssh host00 'which python; python --version'
/home/username/.pyenv/shims/python
Python 3.6.0 :: Anaconda 4.3.1 (64-bit)

$ ssh host01 'which python'
/home/username/.pyenv/shims/python
Python 3.6.0 :: Anaconda 4.3.1 (64-bit)

...
```

Also, the script file (and possibly data files) must be in the same path on each host.

```
$ ls yourscript.py  # in the current directory
yourscript.py

$ ssh host00 "ls $PWD/yourscript.py"
/home/username/your/dir/yourscript.py

$ ssh host01 "ls $PWD/yourscript.py"
/home/username/your/dir/yourscript.py

...
```

If you are using NFS, everything should be okay. If not, you need to transfer all the necessary files manually.

In particular, when you run the ImageNet example in ChainerMN repository, all data files must be available on all computing hosts.

hostfile

The next step is to create a hostfile. A hostfile is a list of hosts on which MPI processes run.:

```
$ vi hostfile
$ cat hostfile
host00
host01
host02
host03
```

Then, you can run your MPI program using the hostfile. To check if the MPI processes run over multiple hosts, save the following script to a file and run it via **mpiexec**:

```
# print_rank.py
import os

from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

for i in range(size):
  if i == rank:
    print("{} {}".format(os.uname()[1], i))
  comm.Barrier()
```

If you get an output like below, it is working correctly.:

```
$ mpiexec -n 4 --hostfile hostfile python print_rank.py
host00 0
host01 1
host02 2
host03 3
```

If you have multiple GPUs, you may want to run multiple processes on each host. You can modify hostfile and specify the number of processes to run on each host.:

```
# If you are using Mvapich:
$ cat hostfile
host00:4
host01:4
host02:4
host03:4

# If you are using Open MPI
$ cat hostfile
host00 cpu=4
host01 cpu=4
host02 cpu=4
host03 cpu=4
```

With this hostfile, try running mpiexec again.:

```
$ mpiexec -n 8 --hostfile hostfile python print_rank.py
host00 0
host00 1
host00 2
host00 3
host01 4
host01 5
host01 6
host01 7
```

You will find that the first 4 processes run on host00 and the latter 4 on host01.

You can also specify computing hosts and resource mapping/binding using command line options of mpiexec. Please refer to the MPI manual for the more advanced use of mpiexec command.

If you get runtime error:

If you get the following error messages, please check the specified section of the troubleshooting or installation guide.

```
[hostxxx:mpi_rank_0][MPIDI_CH3I_SMP_init] CMA is not available. Set MV2_SMP_USE_CMA=0␣
↪to disable CMA.
[cli_0]: aborting job:
Fatal error in PMPI_Init_thread:
Other MPI error, error stack:
MPIR_Init_thread(514)....:
MPID_Init(365)...........: channel initialization failed
MPIDI_CH3_Init(404)......:
MPIDI_CH3I_SMP_Init(2132): process_vm_readv: Operation not permitted


===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 20327 RUNNING AT hostxxx
=   EXIT CODE: 1
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
```

-> Check the value of `MV2_SMP_USE_CMA` (see *CUDA-Aware MPI* and *Check SSH connection and environment variables*).

```
[hostxx:mpi_rank_0][error_sighandler] Caught error: Segmentation fault (signal 11)

===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 20643 RUNNING AT hostxx
=   EXIT CODE: 11
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Segmentation fault (signal 11)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

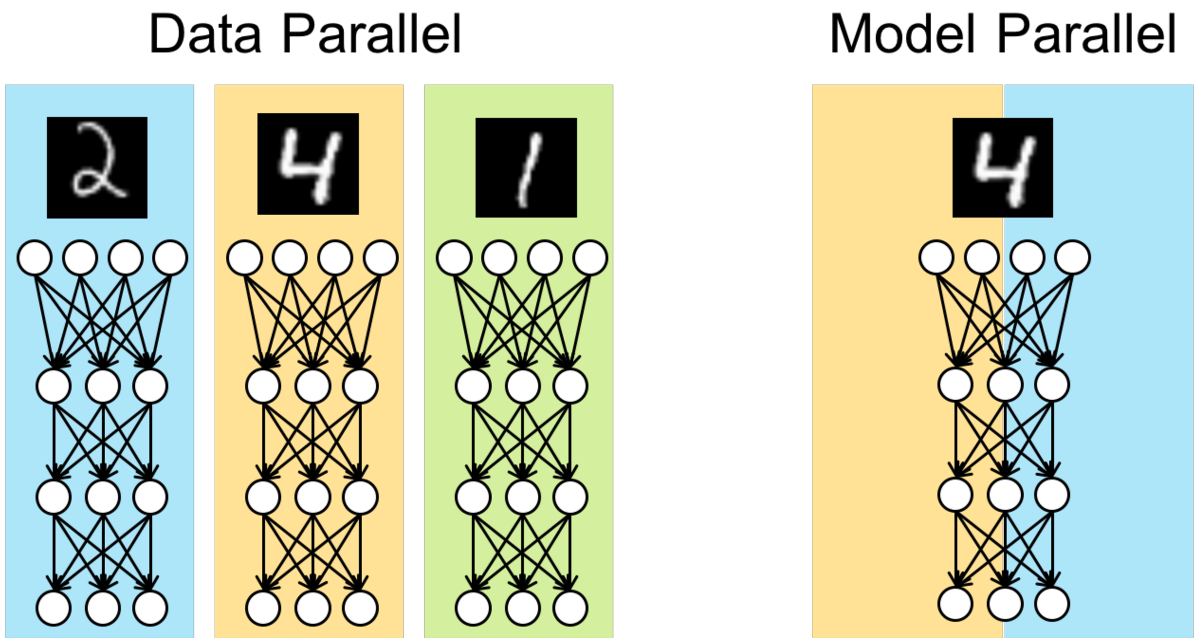
-> Check the value of `MV2_USE_CUDA` (see *CUDA-Aware MPI* and *Check SSH connection and environment variables*)

7.2 Tutorial

7.2.1 Overview

Data Parallelism

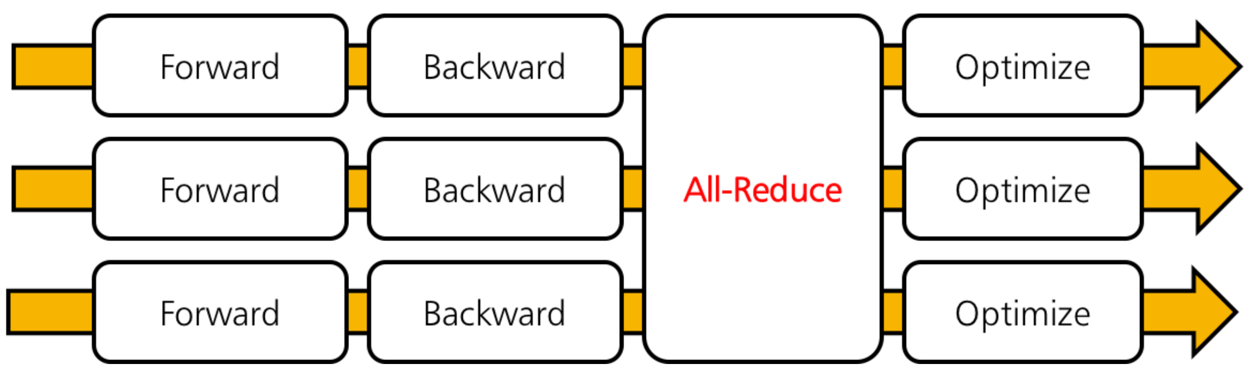
ChainerMN employs the data parallel approach for distributed training. In the data parallel approach, each worker has a model copy, and computes a gradient against a batch. Then, the workers collaborate to update the model using the gradients of all workers.



Training Iterations

What ChainerMN does for distributed training is actually quite simple. Let us look at what we do in each iteration. The following figure illustrates an iteration of standard training using Chainer (without ChainerMN). It consists of three steps: forward, backward and optimize.

When using ChainerMN, an additional step all-reduce is inserted after the backward step. In this step, workers communicate to obtain the averaged gradient over gradients of all workers. Then, the aggregated gradient is used to improve the model in the optimization step.



MPI

ChainerMN is built on MPI. MPI invokes our training script in the SPMD (single program, multiple data) way. ChainerMN is designed to create a process on each GPU. For example, let us suppose you have two nodes with four GPUs each, and want to run train_imagenet.py. Then, you will invoke eight Python processes running train_imagenet.py by using mpiexec or mpirun.

7.2.2 Step 1: Communicators and Optimizers

In the following, we explain how to modify your code using Chainer to enable distributed training with ChainerMN. We take Chainer's MNIST example and modify it in a step-by-step manner to see the standard way of using ChainerMN.

Creating a Communicator

We first need to create a *communicator*. A communicator is in charge of communication between workers. A communicator can be created as follows:

```
comm = chainermn.create_communicator()
```

Workers in a node have to use different GPUs. For this purpose, intra_rank property of communicators is useful. Each worker in a node is assigned a unique intra_rank starting from zero. Therefore, it is often convenient to use the intra_rank-th GPU.

The following line of code is found in the original MNIST example:

```
chainer.cuda.get_device_from_id(args.gpu).use()
```

which we modify as follows:

```
device = comm.intra_rank
chainer.cuda.get_device_from_id(device).use()
```

Creating a Multi-Node Optimizer

This is the most important step. We need to insert the communication right after backprop and right before optimization. In ChainerMN, it is done by creating a *multi-node optimizer*.

Method create_multi_node_optimizer receives a standard Chainer optimizer, and it returns a new optimizer. The returned optimizer is called multi-node optimizer. It behaves exactly same as the supplied original standard optimizer (e.g., you can add hooks such as WeightDecay), except that it communicates model parameters and gradients properly in a multi-node setting.

The following is the code line found in the original MNIST example:

```
optimizer = chainer.optimizers.Adam()
```

To obtain a multi-node optimizer, we modify that part as follows:

```
optimizer = chainermn.create_multi_node_optimizer(
    chainer.optimizers.Adam(), comm)
```

Run

With the above two changes, your script is ready for distributed training. Invoke your script with mpiexec or mpirun (see your MPI's manual for details). The following is an example of executing the training with four processes at localhost:

```
$ mpiexec -n 4 python train_mnist.py
```

In the non-GPU mode, you may see a warning like shown below, but this message is harmless, and you can ignore it for now

```
Warning: using naive communicator because only naive supports CPU-only execution
```

If you have multiple GPUs on the localhost, 4 for example, you may also want to try:

```
$ mpiexec -n 4 python train_mnist.py --gpu
```

Multi-node execution

If you can successfully run the multi-process version of the MNIST example, you are almost ready for multi-node execution. The simplest way is to specify the `--host` argument to the **mpiexec** command. Let's suppose you have two GPU-equipped computing nodes: `host00` and `host01`, each of which has 4 GPUs, and so you have 8 GPUs in total:

```
$ mpiexec -n 8 -host host00,host01 python train_mnist.py
```

The script should print similar results to the previous intra-node execution.

Copying datasets

In the MNIST example, the rank 0 process reads the entire portion of the dataset and scatters it to other processes. In some applications, such as the ImageNet ChainerMN example, however, only the pathes to each data file are scattered and each process reads the actual data files. In such cases, all datasets must be readable on all computing nodes in the same location. You don't need to worry about this if you use NFS (Network File System) or any other similar data synchronizing system. Otherwise, you need to manually copy data files between nodes using **scp** or **rsync**.

If you have trouble

If you have any trouble running the sample programs in your environment, go to the *Step-by-Step Troubleshooting* page and follow the steps to check your environment and configuration.

Next Steps

With only the above two changes distributed training is already performed. Thus, the model parameters are updated by using gradients that are aggregated over all the workers. However, this MNIST example still has a few areas in need of improvment. In the next page, we will see how to address the following problems:

- Training period is wrong; 'one epoch' is not one epoch.
- Evaluation is not parallelized.
- Status outputs to stdout are repeated and annoying.

7.2.3 Step 2: Datasets and Evaluators

Following from the previous step, we continue to explain general steps to modify your code for ChainerMN through the MNIST example. All of the steps below are optional, although useful for many cases.

Scattering Datasets

If you want to keep the definition of 'one epoch' correct, we need to scatter the dataset to all workers.

For this purpose, ChainerMN provides a method `scatter_dataset`. It scatters the dataset of worker 0 (i.e., the worker whose `comm.rank` is 0) to all workers. The given dataset of other workers are ignored. The dataset is split into sub datasets of almost equal sizes and scattered to the workers. To create a sub dataset, `chainer.datasets.SubDataset` is used.

The following line of code from the original MNIST example loads the dataset:

```
train, test = chainer.datasets.get_mnist()
```

We modify it as follows. Only worker 0 loads the dataset, and then it is scattered to all the workers:

```
if comm.rank == 0:
    train, test = chainer.datasets.get_mnist()
else:
    train, test = None, None

train = chainermn.scatter_dataset(train, comm)
test = chainermn.scatter_dataset(test, comm)
```

Creating A Multi-Node Evaluator

This step is also an optional step, but useful when validation is taking a considerable amount of time. In this case, you can also parallelize the validation by using *multi-node evaluators*.

Similarly to multi-node optimizers, you can create a multi-node evaluator from a standard evaluator by using method `create_multi_node_evaluator`. It behaves exactly the same as the given original evaluator except that it reports the average of results over all workers.

The following line from the original MNIST example adds an evaluator extension to the trainer::
trainer.extend(extensions.Evaluator(test_iter, model, device=args.gpu))

To create and use a multi-node evaluator, we modify that part as follows:

```
evaluator = extensions.Evaluator(test_iter, model, device=device)
evaluator = chainermn.create_multi_node_evaluator(evaluator, comm)
trainer.extend(evaluator)
```

Suppressing Unnecessary Extensions

Some of extensions should be invoked only by one of the workers. For example, if the `PrintReport` extension is invoked by all of the workers, many redundant lines will appear in your console. Therefore, it is convenient to register these extensions only at workers of rank zero as follows:

```
if comm.rank == 0:
    trainer.extend(extensions.DumpGraph('main/loss'))
    trainer.extend(extensions.LogReport())
    trainer.extend(extensions.PrintReport(
        ['epoch', 'main/loss', 'validation/main/loss',
         'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))
    trainer.extend(extensions.ProgressBar())
```

7.2.4 Tips and FAQs

Using MultiprocessIterator

If you are using `MultiprocessIterator` and communication goes through InfiniBand, you would probably face crashing problems. This is because `MultiprocessIterator` creates child processes by the `fork` system call, which has incompatibilities with the design of MPI and InfiniBand. To cope with this issue, use `multiprocessing.set_start_method` to start child processes, with a process explicitly forked right after, **before communicator is created** as follows:

```
multiprocessing.set_start_method('forkserver')
p = multiprocessing.Process()
p.start()
p.join()

communicator = chainermn.create_communicator(...)
```

Either `forkserver` mode or `spawn` mode should work. See our ImageNet example script for working sample code of `MultiprocessIterator` and `forkserver`. Unfortunately, `multiprocessing.set_start_method` is only available in Python 3.4+.

Using Your Own Evaluator

Method `create_multi_node_evaluator` can also be used for customized evaluator classes that inherit from `chainer.training.extensions.Evaluator`. Specifically, it wraps the `evaluate` method and returns the averaged values over all workers. Please also refer to our ImageNet example, where a customized evaluator is used.

Using MPI4py Communicator

ChainerMN is based on MPI4py. For advanced users (e.g., those who want to parallelize preprocessing, create custom extension, etc.), we encourage you to make use of MPI4py communicators. Let `comm` be a ChainerMN communicator, then you can obtain MPI4py communicator by `comm.mpi_comm`. Please refer to MPI4py API reference.

Using FP16

FP16 (16-bit half precision floating point values) is supported in `pure_nccl` of a ChainerMN communicator.

MPI process hangs after an unhandled Python exception.

An MPI runtime is expected to kill all of its child processes if one of them exits abnormally or without calling *MPI_Finalize()*. However, when a Python program runs on *mpi4py*, the MPI runtime often fails to detect the process failure, and the rest of the processes hang infinitely. It is especially problematic when you run your ChainerMN program on a cloud environment, in which you are charged on time basis.

This tiny program demonstrates the issue (note that it is not specific to ChainerMN).:

```
# test.py
def func():
  import mpi4py.MPI
  mpi_comm = mpi4py.MPI.COMM_WORLD
  if mpi_comm.rank == 0:
    raise ValueError('failure!')
```

(continues on next page)

(continued from previous page)

```
    mpi4py.MPI.COMM_WORLD.Barrier()

if __name__ == '__main__':
    func()

# mpiexec -n 2 python test.py
```

mpi4py offers a solution to force all processes to abort if an uncaught exception occurs..

```
$ mpiexec -n 2 python -m mpi4py yourscript.py ...
```

This also works well with ChainerMN. See here for more details.

If you cannot apply the solution (i.e. you don't have a control of how Python interpreter is invoked), you can inject the following code snippet into your script file

```
import sys

# === begin code snippet
_old_hook = sys.excepthook

# Global error handler
def global_except_hook(exctype, value, traceback):
    import sys
    try:
        import mpi4py.MPI

$ mpiexec -n 2 -x CHAINERMN_FORCE_ABORT_ON_EXCEPTION=1 python yourscript.py ...
```

Alternatively, you can explicitly call `chainermn.global_except_hook.add_hook()` from your code.:

```
import chainermn

chainermn.global_except_hook.add_hook()
```

The handler hooks uncaught exceptions and call *MPI_Abort()* to ensure that all process are terminated.

You can choose any of these solutions depending on your environment and restrictions.

NOTE: These techniques are effective only for unhandled Python exceptions. If your program crashes due to lower-level issues such as *SIGSEGV*, the MPI process may still hang.

7.3 Model Parallel

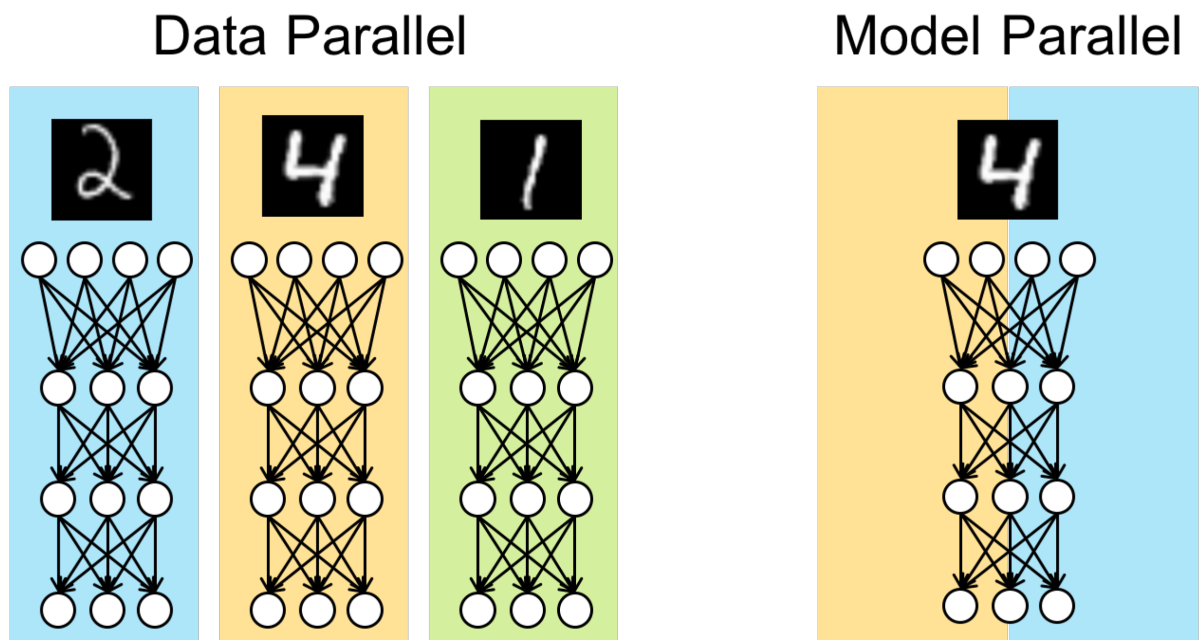
7.3.1 Overview

Model Parallelism

Even though ChainerMN mainly supports the data parallel approach for distributed training, it also has experimental APIs for the *model parallel* approach. The model parallel approach splits a given model into subcomponents loaded on several processes. This approach is useful in cases where

- large mini-batch or high-resolution is needed.
- the model is too huge to run on a single process.

- the mixture of experts are trained.



Philosophy

ChainerMN takes the following three approaches to realize the model parallelism.

1. Communication as Function

ChainerMN provides several special functions for communications such as `chainermn.functions.bcast` and `chainermn.functions.alltoall`, which wraps raw MPI communications. Users define communications between processes as Chainer function calls in the model definitions. This enables highly flexible communication patterns. Moreover, parameter updates in backward propagation are automatically invoked through `backward` defined in those functions for communications.

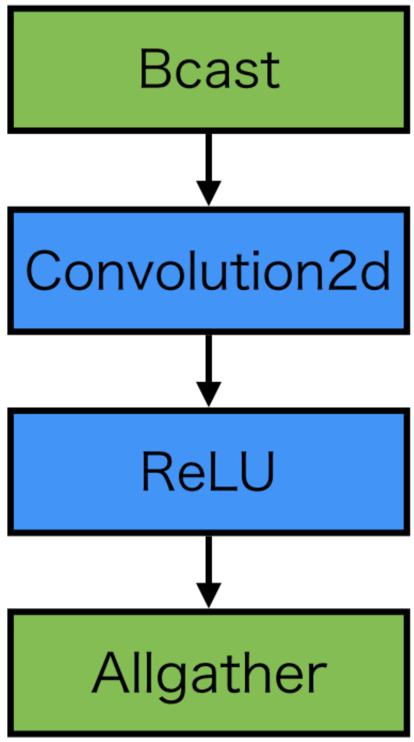
2. Synchronous Model Parallel

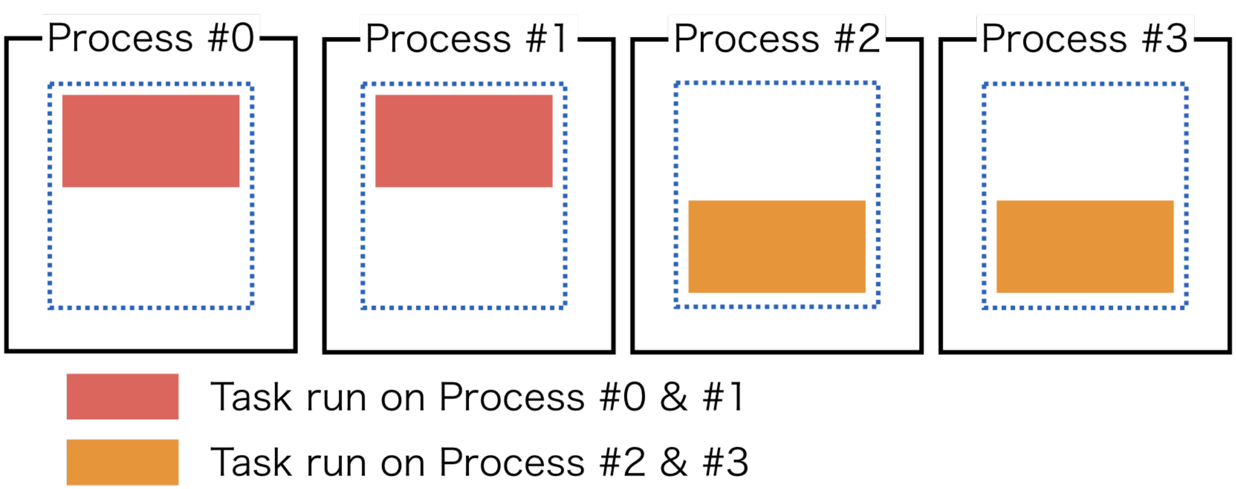
ChainerMN restricts itself to synchronous SGD. Though the asynchronous counterpart seems to be more computationally efficient, asynchronous SGD often suffer from the stale gradients problem and results in difficulty while debugging. ChainerMN's synchronous communication model makes SGD simpler.

3. Single-Program-Multiple-Data (SPMD)

In principle, ChainerMN supports single-program-multiple-data (SPMD), which means the same program is invoked and different data are used on each process.

Synchronous model-parallelism suits well with MPI programming style and SPMD model.


Bcast
Convolution2d
ReLU
Allgather


Process #0
Process #1
Process #2
Process #3
Task run on Process #0 & #1
Task run on Process #2 & #3

References

- More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server
- Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer
- AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks
- Deep Mixture of Experts via Shallow Embedding
- Mesh-TensorFlow: Deep Learning for Supercomputers
- GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

7.3.2 Model Parallel on ChainerMN

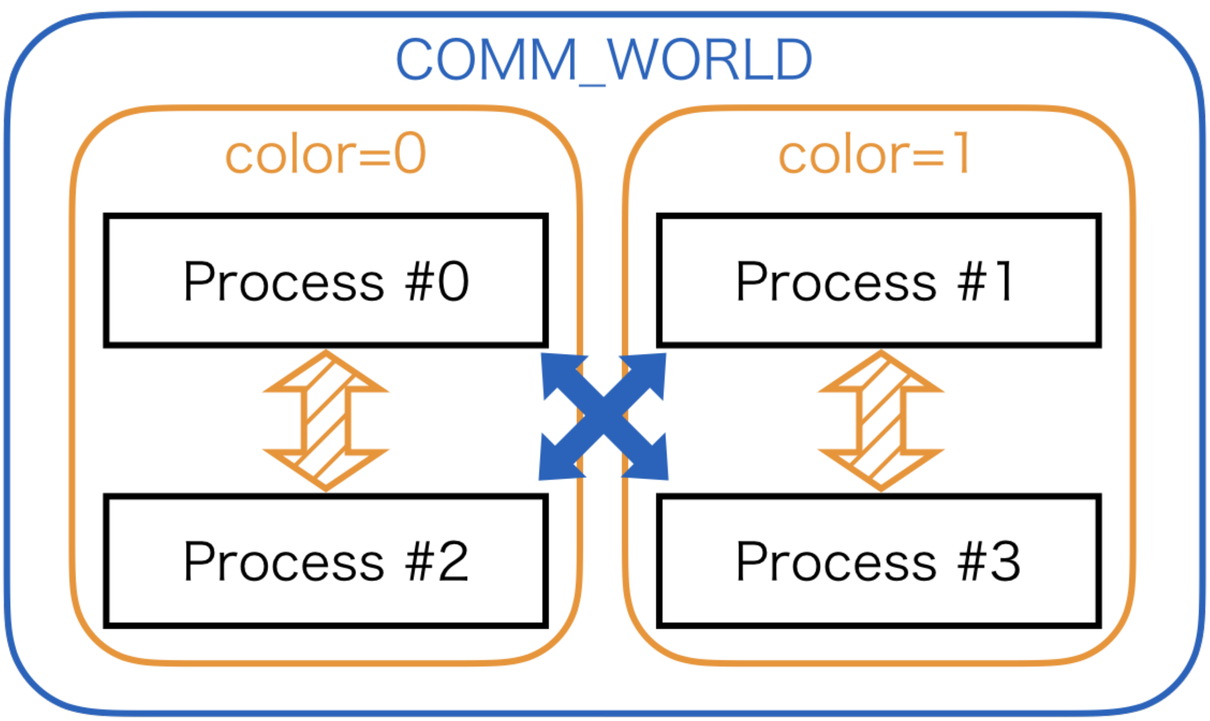
Step 1: Communicators

To perform multi-node communications, a *communicator* is needed. Basic usages are the same with the case of the data parallel, see *Step 1: Communicators and Optimizers*:

```
comm = chainermn.create_communicator()
```

If you want to define collective communications among limited number of processes later, it is useful to split the communicator:

```
subcomm = comm.split(comm.rank % 2, comm.rank)
```



For further detail about the communicator split, please refer to MPI tutorial.

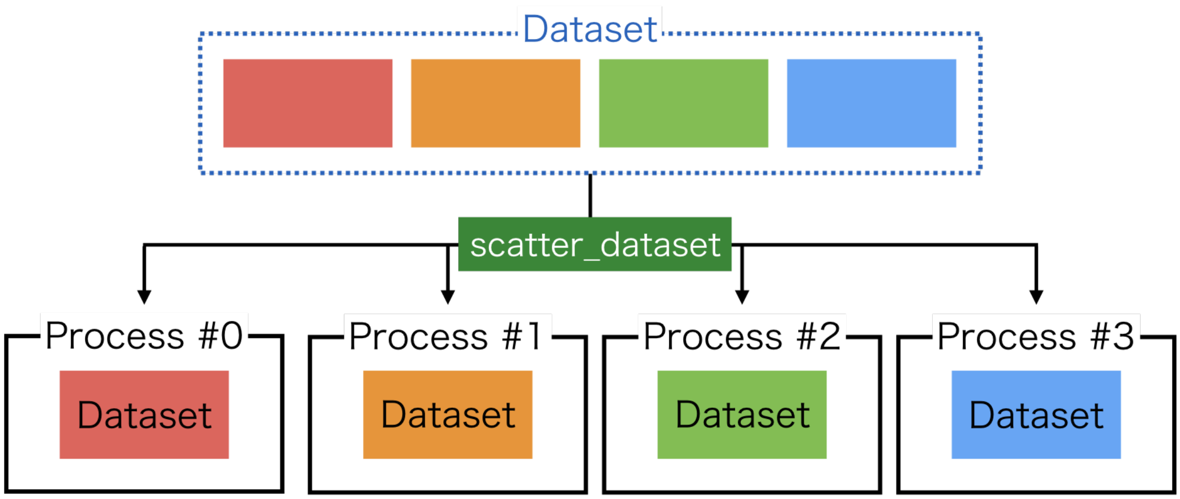
Step 2: Datasets and Iterators

In model parallel training, all processes belong to at least one of the following dataset input patterns.

1. model inputs come from datasets, and each process takes different mini-batches
2. model inputs come from datasets, and several processes share the same mini-batches
3. model inputs come from other processes

1. scatter_dataset

For the first case, you may use `scatter_dataset` as is introduced in *Step 2: Datasets and Evaluators*.



2. multi node iterator

For the second case, iterator need to be modified, where `create_multi_node_iterator` is useful:

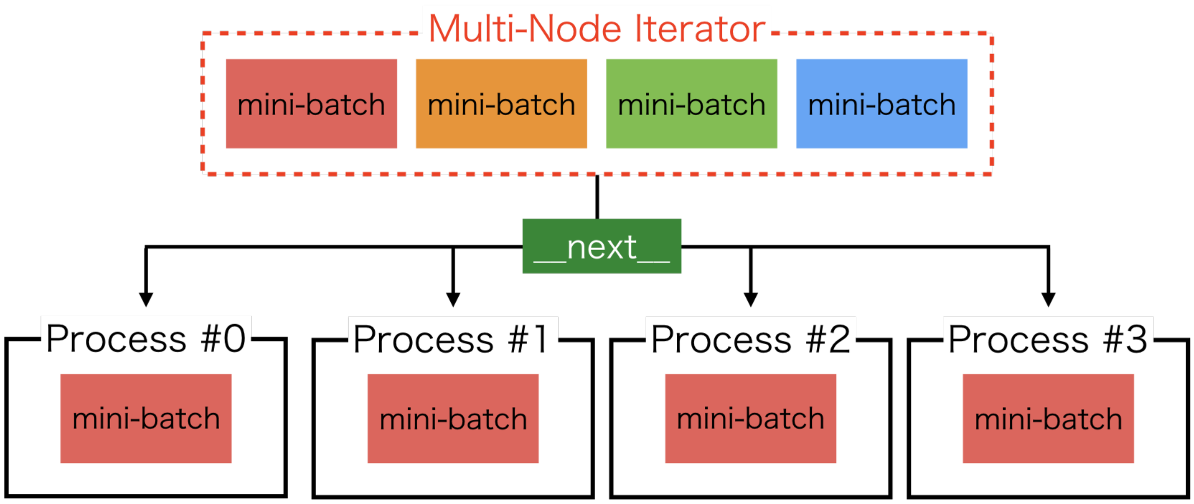
```
train, test = chainer.datasets.get_mnist()
train_iter = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(train, batchsize), comm)
test_iter = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(test, batchsize), comm)
```

The resulting iterators return the same mini-batches among processes specified by the communicator.

3. empty dataset

For the last case, you may use `create_empty_dataset`, which returns a dataset with the same number of empty tuples as the original dataset:

```
train, test = chainer.datasets.get_mnist()
train = chainermn.datasets.create_empty_dataset(train)
test = chainermn.datasets.create_empty_dataset(test)
```



This input pattern appears in the subsequent examples such as *Example 1: Simple MLP*. Note that datasets are required in Chainer's updater API. The empty dataset can be used as a dummy dataset.

Step 3: Define Communications

ChainerMN supports most of the MPI communications *as Chainer functions*, including point-to-point and collective communications. To know usages of each communication, please refer to *API Reference*.

Example 1: Point-to-point Communication

This is an example to use point-to-point communications:

```
def __call__(self, x):
    h = f(x)
    h = chainermn.functions.send(x, comm, rank=1)
    return h
```

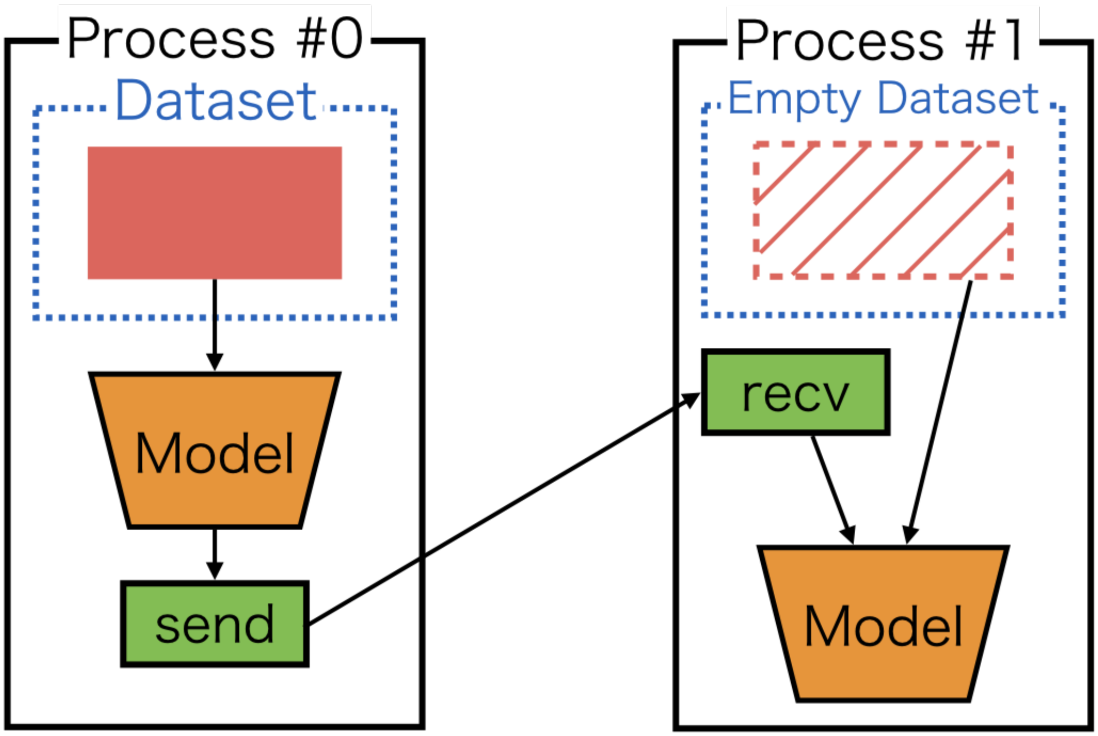
The communication target is specified by `rank` parameter. Note that the return value of `send` is often not negligible. Please refer to *Note: Define-by-Run and Model Parallelism*.

Example 2: Collective Communication

Here is another example to use collective communications:

```
def __call__(self, x):
    h = f(x)
    h = chainermn.functions.allgather(comm, h)
    h = F.stack(h, axis=0)
    h = F.average(h, axis=0)
    return h
```

This pattern often appears in the averaging ensemble training.



Note: Define-by-Run and Model Parallelism

In model-parallel training, a model on each process may become *non-connected* computational graph. Let's take a look at an example.

Naive implementation of a model on process #0 could be:

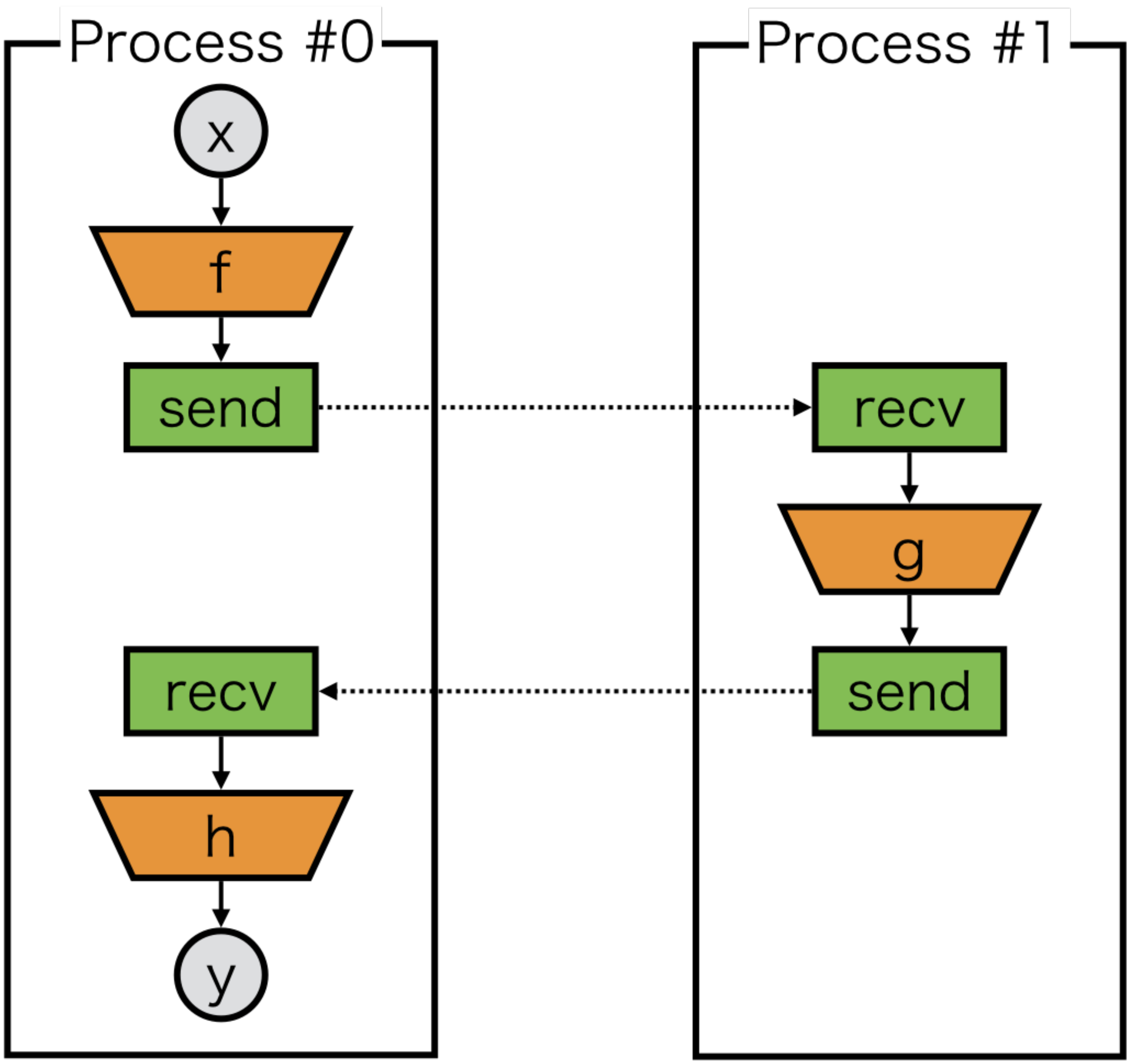
```
class Model_0(chainer.Chain):
    def __call__(self, x):
        # first component
        z = f(x)
        chainermn.functions.send(z, comm, rank=1)

        # second component
        z = chainermn.functions.recv(comm, rank=1)
        y = h(z)

        return y
```

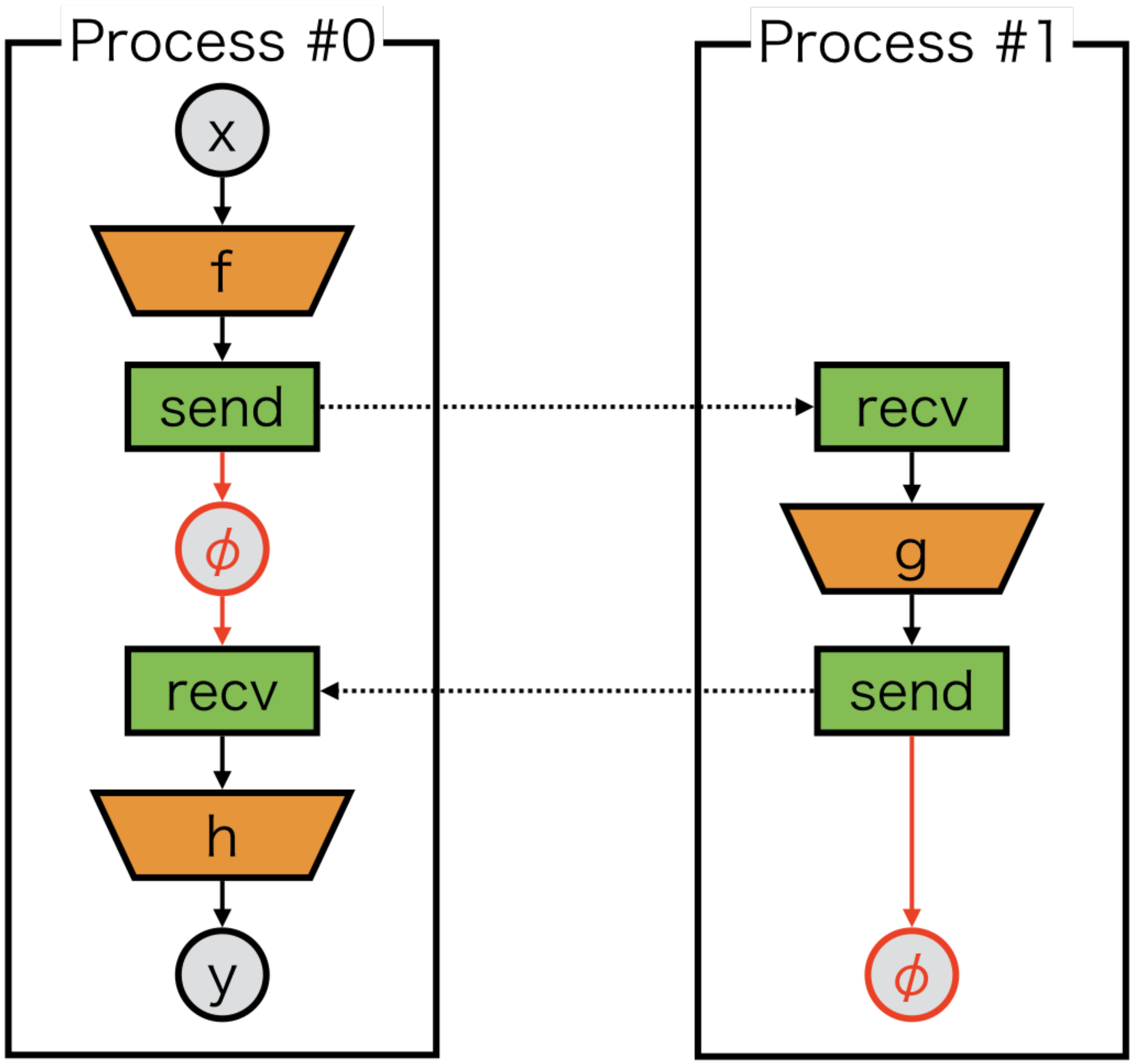
One may notice that there is no connection between the first and second components of computational graph. As we rely on defined-by-run framework, we cannot build a backward path from the second component to the first component. In order to build the backward path, a dummy variable, which we call `delegate_variable`, is needed.

The variable ϕ in the above figure is `delegate_variable`, which is a return value of `send` and passed to an argument of `recv`:

```
class Model_0(chainer.Chain):
    def __call__(self, x):
        # first component
```

(continues on next page)


Process #0
x
f
send
recv
h
y
Process #1
recv
g
send


Process #0
Process #1
x
f
send
recv
ϕ
recv
g
send
h
y
ϕ

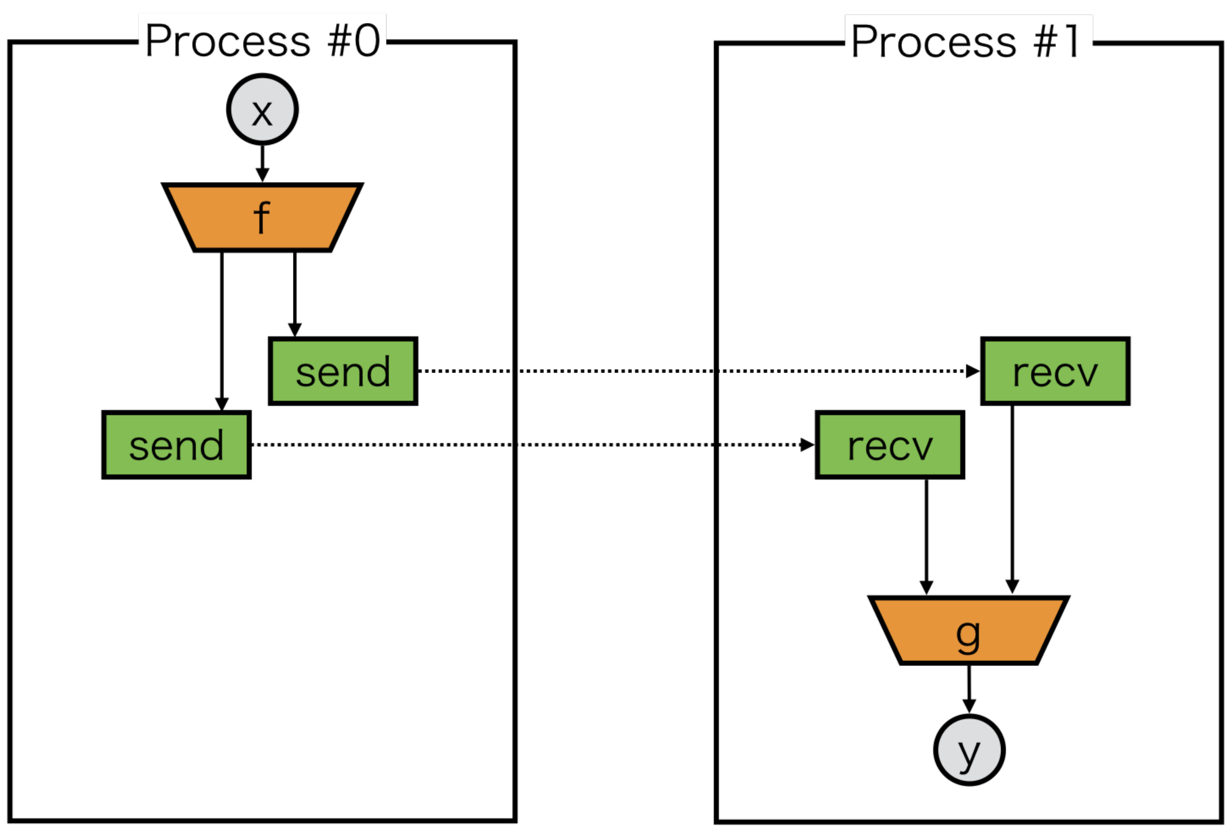
(continued from previous page)

```
        z = f(x)
        phi = chainermn.functions.send(z, comm, rank=1)

        # second component
        z = chainermn.functions.recv(comm, rank=1, delegate_variable=phi)
        y = h(z)

        return y

class Model_1(chainer.Chain):
    def __call__(self, _):
        z = chainermn.functions.recv(comm, rank=0)
        z = g(z)
        phi = chainermn.functions.send(z, comm, rank=0)
        return phi
```

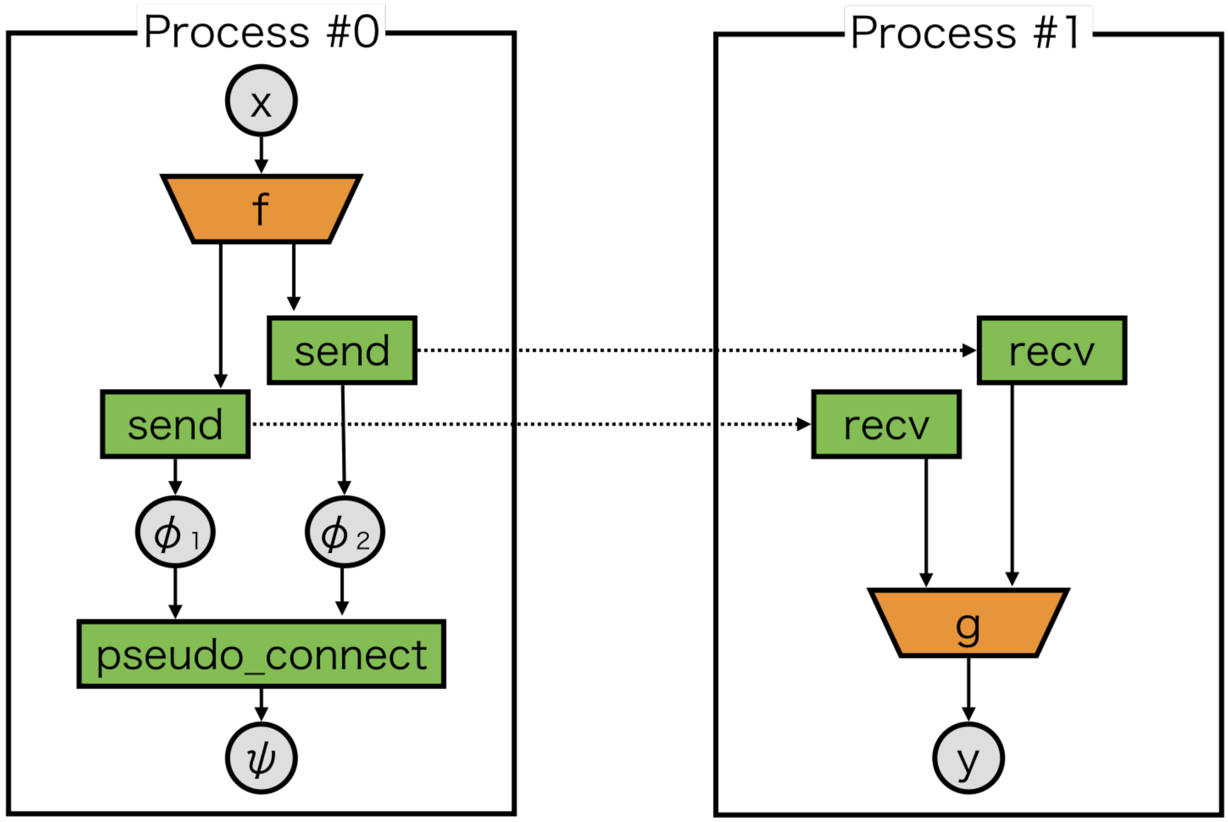
`Model_1` also need to return a delegate variable ϕ to backtrack its computational graph to compute gradients. Thus, the backward computation is guaranteed. **Otherwise, backward computation will cause deadlock**.

Note: Delegate Variable and Pseudo Connect

As we just see above, delegate variables must be appropriately handled to avoid potential deadlock. However, there are still some pathological cases. Let's consider to `send` variables twice.



Here, we must guarantee that backward tracking can find two `send`, but we can only return one delegate variable from each model. `pseudo_connect` is a special function to combine one delegate variable to another variable.



In the above case, the returned variable ψ from `pseudo_connect` behaves as if it is ϕ_2, while its `backward` backtracks both ϕ_1 and ϕ_2:

```
class Model_0(chainer.Chain):
    def __call__(self, x):
        z1, z2 = f(x)
        phi1 = chainermn.functions.send(z1, comm, rank=1)
        phi2 = chainermn.functions.send(z2, comm, rank=1)
        psi = chainermn.functions.pseudo_connect(phi1, phi2)
        return psi

class Model_1(chainer.Chain):
    def __call__(self, _):
        z1 = chainermn.functions.recv(comm, rank=0)
        z2 = chainermn.functions.recv(comm, rank=0)
        y = g(z1, z2)
        return y
```

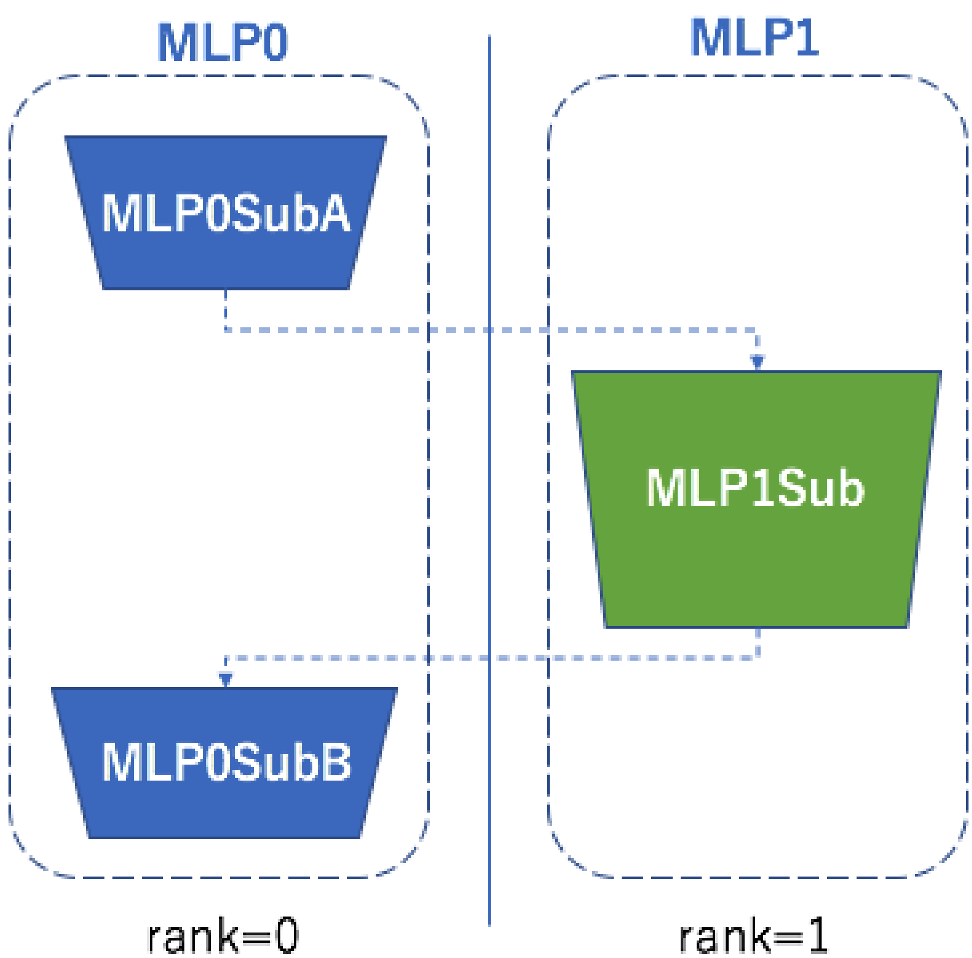
7.3.3 Example 1: Simple MLP

Here is the first example of model parallel, a simple MLP separated on two processes.

First, let's create a ChainerMN communicator:

```
if args.gpu:
    comm = chainermn.create_communicator('pure_nccl')
```

(continues on next page)


MLP0
MLP1
MLP0SubA
MLP1Sub
MLP0SubB
rank=0
rank=1

(continued from previous page)

```
    device = comm.intra_rank
else:
    comm = chainermn.create_communicator('naive')
    device = -1
```

As we saw in *Model Parallel on ChainerMN*, one naive implementation would be to use the point-to-point communication such as `send` and `recv`:

```
class MLP0(chainer.Chain):
    def __init__(self, comm, n_out):
        super(MLP0SubA, self).__init__(
            l1=L.Linear(784, n_out))

    def __call__(self, x):
        h0 = F.relu(self.l1(x))
        phi = chainermn.functions.send(h0, self.comm, rank=1)
        # Note: do not forget to pass delegate variable
        y = chainermn.functions.recv(self.comm, rank=1, delegate_variable=phi)
        return y

class MLP1(chainer.Chain):
    def __init__(self, n_units, n_out):
        super(MLP1Sub, self).__init__(
            l2=L.Linear(None, n_units),
            l3=L.Linear(None, n_out))

    def __call__(self, _):
        h0 = chainermn.functions.recv(self.comm, rank=0)
        h1 = F.relu(self.l2(h0))
        return chainermn.functions.send(self.l3(h1), self.comm, rank=0)
```

One should note that

- `MLP0`: delegate variable is indispensable which is passed from `send` to `recv`.
- `MLP1`: the return value from `send` must be returned in `__call__`, which is used to track back the computational graph.

On each process, different models are trained:

```
if comm.rank == 0:
    model = L.Classifier(MLP0(comm, 100))
elif comm.rank == 1:
    model = MLP1(comm, 100, 10)
```

Since `MLP1` receives its inputs from `MLP0` over the point-to-point communication, let's use `empty_dataset` instead of the usual dataset:

```
# Iterate dataset only on worker 0.
train, test = chainer.datasets.get_mnist()
if comm.rank == 1:
    train = chainermn.datasets.create_empty_dataset(train)
    test = chainermn.datasets.create_empty_dataset(test)
```

Now we can run a model parallel architecture.

There is an alternative API to define the same model without explicitly defining communication paths:

```
class MLP0SubA(chainer.Chain):
    def __init__(self, comm, n_out):
        super(MLP0SubA, self).__init__(
            l1=L.Linear(784, n_out))

    def __call__(self, x):
        return F.relu(self.l1(x))

class MLP0SubB(chainer.Chain):
    def __init__(self, comm):
        super(MLP0SubB, self).__init__()

    def __call__(self, y):
        return y

class MLP0(chainermn.MultiNodeChainList):
    # Model on worker 0.
    def __init__(self, comm, n_out):
        super(MLP0, self).__init__(comm=comm)
        self.add_link(MLP0SubA(comm, n_out), rank_in=None, rank_out=1)
        self.add_link(MLP0SubB(comm), rank_in=1, rank_out=None)

class MLP1Sub(chainer.Chain):
    def __init__(self, n_units, n_out):
        super(MLP1Sub, self).__init__(
            l2=L.Linear(None, n_units),
            l3=L.Linear(None, n_out))

    def __call__(self, h0):
        h1 = F.relu(self.l2(h0))
        return self.l3(h1)

class MLP1(chainermn.MultiNodeChainList):
    # Model on worker 1.
    def __init__(self, comm, n_units, n_out):
        super(MLP1, self).__init__(comm=comm)
        self.add_link(MLP1Sub(n_units, n_out), rank_in=0, rank_out=0)
```

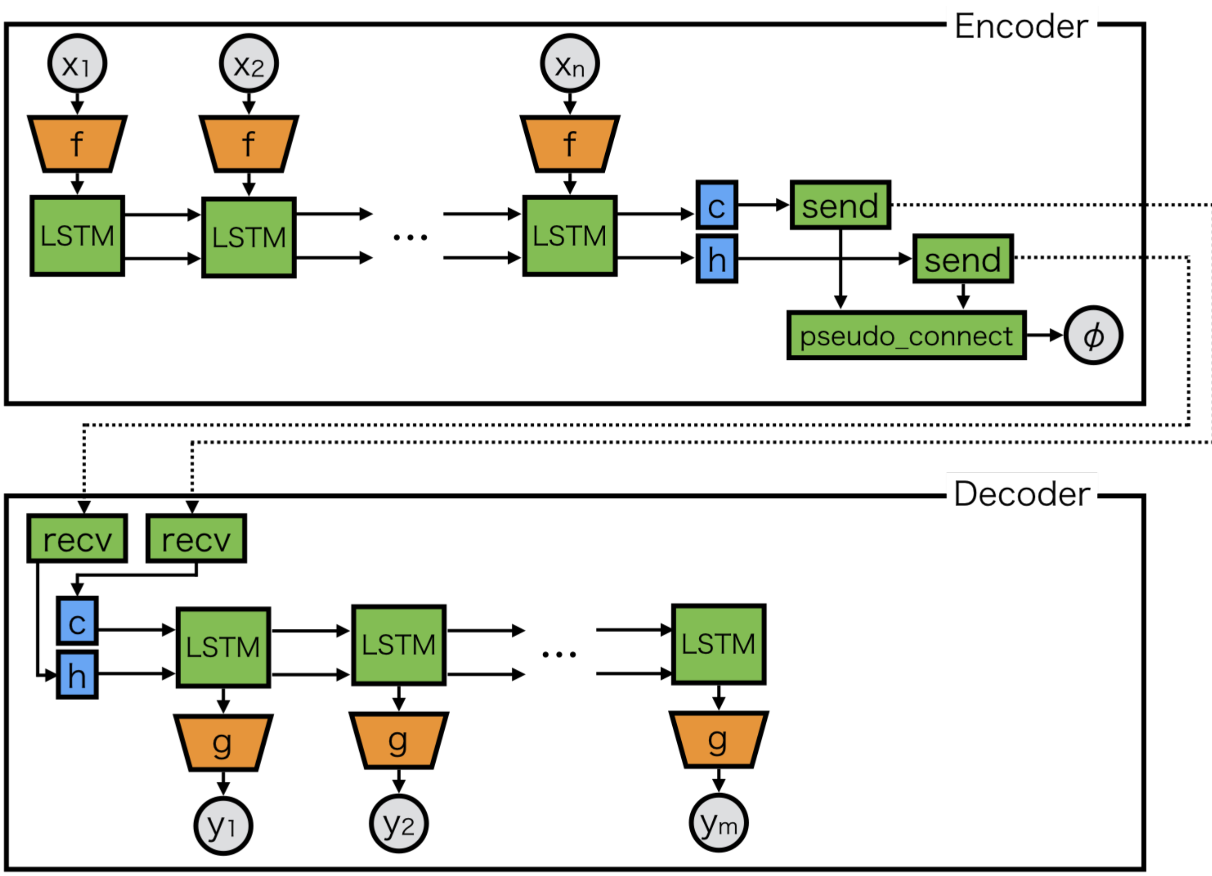
`MultiNodeChainList` enables to define a multi model architecture, by adding non-connected component with `add_link`. Two arguments `rank_in` and `rank_out` specifies from which process the added link receives their inputs, and to which process it sends their outputs.

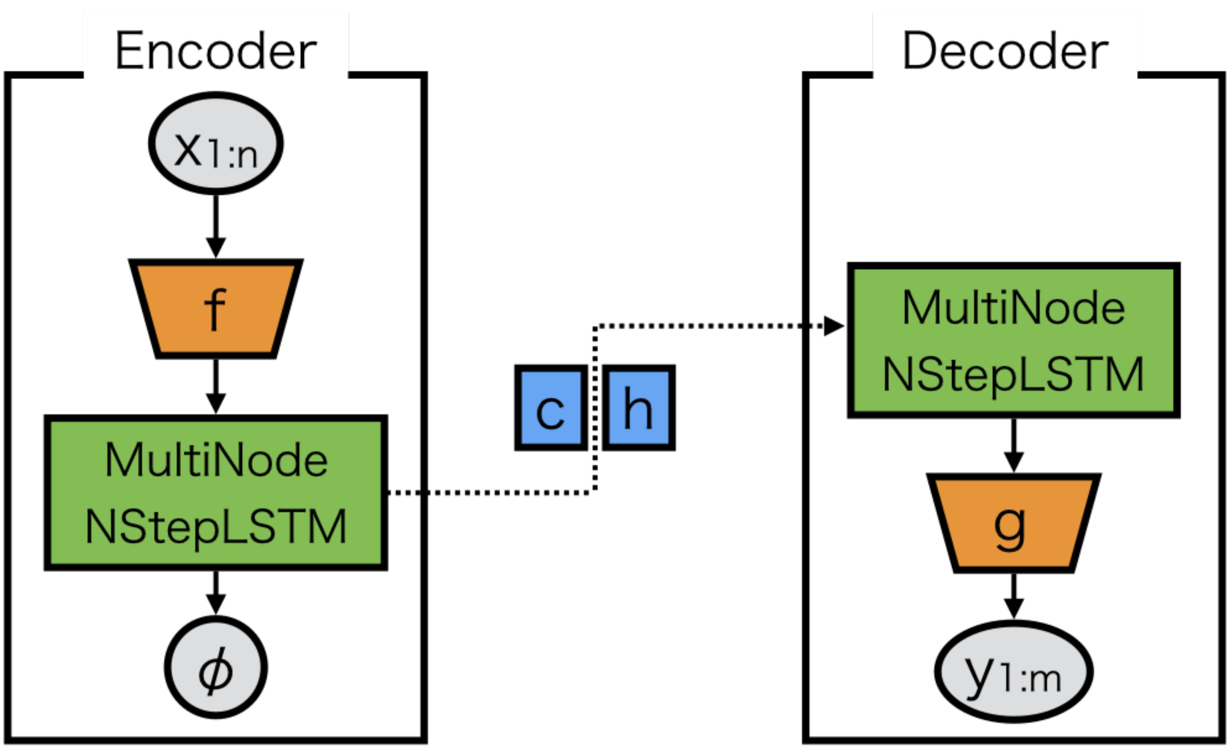
Although it may seems that there is no necessity to parallelize MLP with this size, it can be useful to train a MLP with many layers and parameters so that the entire model cannot be loaded on a single GPU. The entire training code is available here.

7.3.4 Example 2: seq2seq

This example shows how to parallelize models that involves RNN.

Above figure depicts a typical encoder-decoder model, where the model is split up to encoder and decoder, both running respectively in two processes. When `f` or `g` are large models that consume huge memory such as CNN, model parallelism like this would be useful. In the forward computation, the encoder invokes `send` function to send its context vectors, and the decoder invokes `recv` to receive them. The backward computation must be built by `pseudo_connect`. As this communication pattern is very popular in RNNs, `MultiNodeNStepRNN` is a ready-made utility link for this pattern. It can replace this complicated communication pattern.


Encoder
x1
x2
xn
f
f
f
LSTM
LSTM
LSTM
...
c
h
send
send
pseudo_connect
ϕ
Decoder
recv
recv
c
h
LSTM
LSTM
LSTM
...
g
g
g
y1
y2
ym


Encoder
x1:n
f
MultiNode
NStepLSTM
ϕ
c
h
Decoder
MultiNode
NStepLSTM
g
y1:m

MultiNodeNStepRNN can be created by create_multi_node_n_step_rnn:

```
rnn = chainermn.links.create_multi_node_n_step_rnn(
    L.NStepLSTM(n_layers, n_units, n_units, 0.1),
    comm, rank_in=None, rank_out=1)
```

where comm is a ChainerMN communicator (see *Step 1: Communicators*).

The overall model definition can be written as follows:

```
class Encoder(chainer.Chain):

    def __init__(self, comm, n_layers, n_units):
        super(Encoder, self).__init__(
            # Corresponding decoder LSTM will be invoked on process 1.
            mn_encoder=chainermn.links.create_multi_node_n_step_rnn(
                L.NStepLSTM(n_layers, n_units, n_units, 0.1),
                comm, rank_in=None, rank_out=1
            ),
        )
        self.comm = comm
        self.n_layers = n_layers
        self.n_units = n_units

    def __call__(self, *xs):
        exs = f(xs)
        c, h, _, phi = self.mn_encoder(exs)
        return phi

class Decoder(chainer.Chain):

    def __init__(self, comm, n_layers, n_units):
        super(Decoder, self).__init__(
            # Corresponding encoder LSTM will be invoked on process 0.
            mn_decoder=chainermn.links.create_multi_node_n_step_rnn(
                L.NStepLSTM(n_layers, n_units, n_units, 0.1),
                comm, rank_in=0, rank_out=None),
        )
        self.comm = comm
        self.n_layers = n_layers
        self.n_units = n_units

    def __call__(self, *ys):
        c, h, os, _ = self.mn_decoder(ys)
        # compute loss (omitted)
```

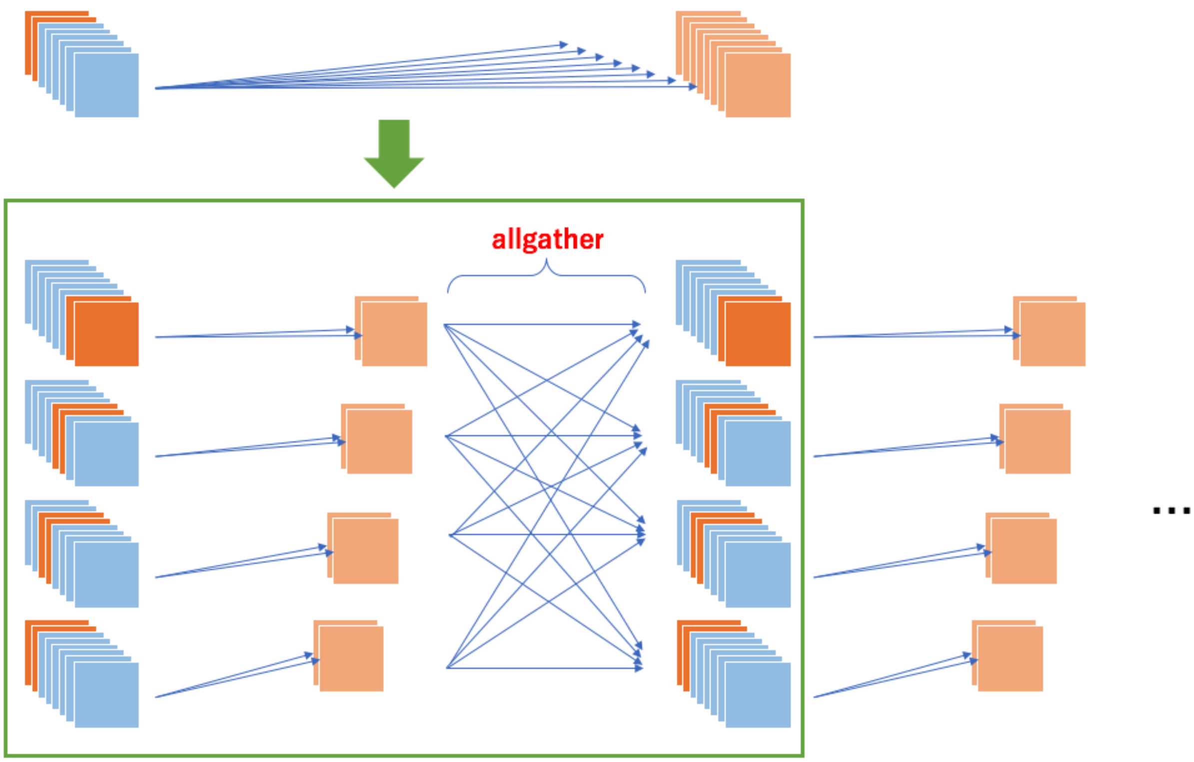
An example code with a training script is available here.

7.3.5 Example 3: Channel-wise Parallel Convolution

This is an example to parallelize CNN in channel-wise manner. This parallelization is useful with large batch size, or with high resolution images.

The basic strategy is

1. to pick channels that each process is responsible for
2. to apply convolution, and



3. to use `allgather` to combine outputs of all channels into a single tensor

on each process. Parallel convolution model implementation could be like this:

```
class ParallelConvolution2D(chainer.links.Convolution2D):
    def __init__(self, comm, in_channels, out_channels, *args, **kwargs):
        self.comm = comm
        self.in_channels = in_channels
        self.out_channels = out_channels
        super(ParallelConvolution2D, self).__init__(
            self._in_channel_size, self._out_channel_size, *args, **kwargs)

    def __call__(self, x):
        x = x[:, self._channel_indices, :, :]
        y = super(ParallelConvolution2D, self).__call__(x)
        ys = chainermn.functions.allgather(self.comm, y)
        return F.concat(ys, axis=1)

    def _channel_size(self, n_channel):
        # Return the size of the corresponding channels.
        n_proc = self.comm.size
        i_proc = self.comm.rank
        return n_channel // n_proc + (1 if i_proc < n_channel % n_proc else 0)

    @property
    def _in_channel_size(self):
        return self._channel_size(self.in_channels)

    @property
    def _out_channel_size(self):
```

(continues on next page)

(continued from previous page)

```
        return self._channel_size(self.out_channels)

    @property
    def _channel_indices(self):
        # Return the indices of the corresponding channel.
        indices = np.arange(self.in_channels)
        indices = indices[indices % self.comm.size == 0] + self.comm.rank
        return [i for i in indices if i < self.in_channels]
```

where `comm` is a ChainerMN communicator (see *Step 1: Communicators*).

`ParallelConvolution2D` can simply replace with the original `Convolution2D`. For the first convolution layer, all processes must input the same images to the model. `MultiNodeIterator` distributes the same batches to all processes every iteration:

```
if comm.rank != 0:
    train = chainermn.datasets.create_empty_dataset(train)
    test = chainermn.datasets.create_empty_dataset(test)

train_iter = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(train, args.batchsize), comm)
test_iter = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(test, args.batchsize,
                                     repeat=False, shuffle=False),
    comm)
```

An example code with a training script for VGG16 parallelization is available here.

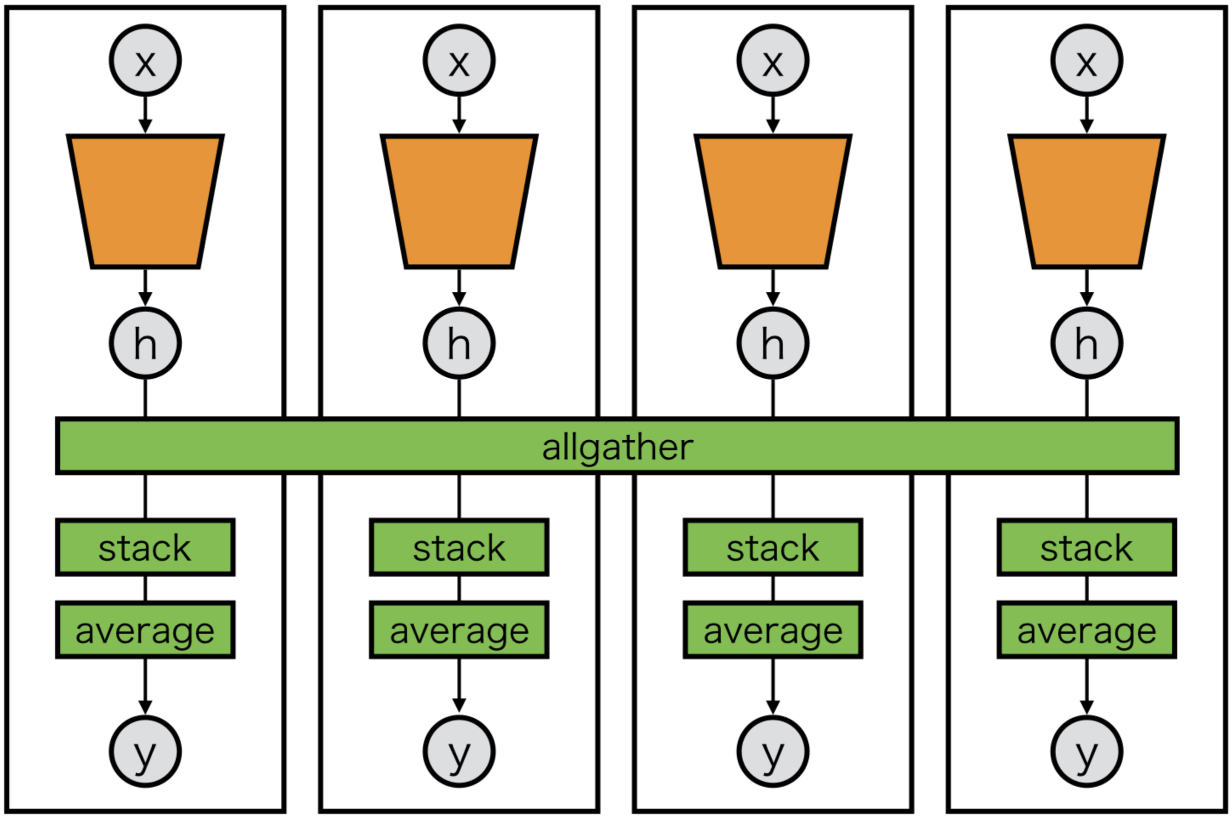
7.3.6 Example 4: Ensemble

Ensemble is a training technique to obtain better classification performance by combining multiple base classifiers. Averaging ensemble is one of the simplest examples of ensemble, which takes average of all classifier outputs in the test phase. Model parallelism and collective communications can effectively help to implement it.

The following wrapper makes model parallel averaging ensemble easier:

```
class Averaging(chainer.Chain):
    def __init__(self, comm, block):
        super(Averaging, self).__init__()
        self.comm = comm
        with self.init_scope():
            self.block = block

    def __call__(self, x):
        y = self.block(x)

        if not chainer.config.train:
            y = chainermn.functions.allgather(self.comm, y)
            y = F.stack(y, axis=0)
            y = F.average(y, axis=0)

        return y
```

Then, any links wrapped by `Averaging` are ready to be parallelized and averaged:




```
class Model(chainer.Chain):
    def __init__(self, comm):
        super(Model, self).__init__()
        self.comm = comm
        with self.init_scope():
            self.l1 = L.Linear(d0, d1)
            self.l2 = L.Linear(d1, d2)
            self.l3 = Averaging(self.comm, L.Linear(d2, d3))

    def __call__(self, x):
        h = F.relu(self.l1(x))
        h = F.relu(self.l2(h))
        y = F.relu(self.l3(h))
        return y
```

From the perspective of model inputs/outputs, the averaged model is compatible with the original model. Thus, we only need to replace the last layer with the averaged layer.

In averaging ensemble, each base classifier is trained independently and ensembled in the test phase. This can be implemented by using `MultiNodeIterator` only for the test iterator:

```
# train = (training dataset)
# test = (test dataset)

if comm.rank != 0:
    train = chainermn.datasets.create_empty_dataset(train)
    test = chainermn.datasets.create_empty_dataset(test)
```

(continues on next page)

(continued from previous page)

```
train_iter = chainer.iterators.SerialIterator(train, batchsize)
test_iter = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(test, batchsize,
                                     repeat=False, shuffle=False),
    comm)
```

7.4 API Reference

7.4.1 Communicators

chainermn.**create_communicator**(*communicator_name='pure_nccl'*, *mpi_comm=None*, ***kwargs*)

Create a ChainerMN communicator.

Different communicators provide different approaches of communication, so they have different performance charasteristics. The default communicator `pure_nccl` is expected to generally perform well on a variety of environments, so one need not to change communicators in most cases. However, you may need to choose other communicators depending on your computing platform and the availability of NCCL library. The following communicators are available.

Name	CPU	GPU	NCCL	Recommended Use Cases
pure_nccl		OK	Required (>= v2)	`pure_nccl` is recommended when NCCL2 is available in the environment.
flat		OK		N/A
naive	OK	OK		Testing on CPU mode

pure_nccl communicator supports multiple data types, FP32 and FP16, in gradient exchange. The communication data type is determined based on *chainer.global_config.dtype* and *allreduce_grad_dtype*. When *allreduce_grad_dtype* is the default value *None*, FP32 is used when *chainer.global_config.dtype* is *numpy.float32* and FP16 otherwise. *allreduce_grad_dtype* parameter, which is either *numpy.float16* or *numpy.float32*, overwrites the *chainer.global_config.dtype*.

The table blow summarizes the data type selection in gradient exchange.

	allreduce_grad_dtype		
global_config.dtype	None	numpy.float16	numpy.float32
chainer.mixed16	FP16	FP16	FP32
numpy.float16	FP16	FP16	FP32
numpy.float32	FP32	FP16	FP32

Other communicators, namely `flat` and `naive`, support only float32 communication, no matter what the model is. This is due to MPI's limited support of float16.

Parameters

- **communicator_name** – The name of communicator (`naive`, `flat`, or `pure_nccl`)
- **mpi_comm** – MPI4py communicator
- **allreduce_grad_dtype** – Data type of gradient used in All-Reduce. If `None`, the dtype of a model is used.

Returns ChainerMN communicator that implements methods defined in *chainermn.CommunicatorBase*

class chainermn.**CommunicatorBase**

Interface definition of all communicators.

All communicators that have compatible set of methods with this class is supposed to work in ChainerMN's parallel computation implementation. The methods are named after MPI functions, such as `bcast()` came from `MPI_Bcast()`.

There are two types of methods: one that treats Python objects have `_obj` suffix. The other has methods without any suffix and it handles ndarray and arrays filled with scaler values. So the number of methods would be

```
[send, recv, bcast, gather, allreduce] * [ '_obj', '']
```

(with single exception `alltoall`, `multi_node_mean_grad`, `split` and `bcast_data` so far). Also methods are supposed to be written in this order. All those methods must be implemented in its implementation class, or otherwise it cannot be instantiated in runtime.

Note: As most implementation of `_obj`-sufficed methods involves Python object pickling and unpickling, there is an implicit size limit.

TODO(kuenishi): as of now no implementation class actually has `allreduce` method.

abstract allgather(*x*)

A primitive of inter-process all-gather communication.

This method tries to invoke all-gather communication within the communicator. All processes in the communicator are expected to invoke `allgather()`. This method relies on mpi4py fast communication optimized for numpy arrays, as well as `send()` and `recv()`.

Note that this method can only handle the same shapes of data over all processes, and cannot handle tuple data.

Parameters **x** (*numpy/cupy array*) – Array to be gathered.

Returns Received arrays.

Return type ys (tuple of numpy/cupy array)

abstract allreduce(*data*)

Allreduce operation among processes

Processes one of several aggregation operations using all data from all processes and returns the result of the aggregation to all processes.

TODO(kuenishi): add `op` argument once we find a use case for operations other than 'SUM'.

Parameters **data** (`ndarray`) – the data to aggregate among nodes.

Returns Sum of all data from all processes.

allreduce_grad(*model*, *zero_fill=False*)

mean Chainer model gradients.

Deprecated since version v7.0.0: This API is deprecated. Please use *multi_node_mean_grad()* instead.

Parameters

- **link** (`Link`) – Link object.

- **zero_fill** – A knob to control whether to fill gradients of initialized and unused Link (which is None internally) with zero-valued array, because the all gradients must be an array among processes for performing all-reduce, which might be an array or None after backward computation. Gradients of uninitialized Link are skipped. If it is False, gradients of unused Link are just skipped.

abstract allreduce_obj(*obj*)

Apply a reduce operation to all objects and spread the result.

For example of integers and summation, equivalent local code is:

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

The only operation currently supported is summation.

TODO(kuenishi): support other operations such as 'MAX', 'MIN' and 'PROD' with `op` argument once we need any of them.

Parameters obj – An arbitrary object to apply reduce operation. Must have corresponding operation method e.g. `__plus__()`.

Returns The result of the operation applied to all objects.

abstract alltoall(*xs*)

All-to-all implementation for ndarray

Parameters xs (*tuple of numpy/cupy array*) –

Returns Received arrays. The length of tuple equals to the communicator size.

Return type ys (tuple of numpy/cupy array)

abstract bcast(*data*, *max_buf_len=None*, *root=0*)

Broadcasts an ndarray from root process to all processes

Parameters

- **data** (*numpy/cupy array*) – for root process, the data to broadcast. For non-root processes, this argument is ignored.
- **max_buf_len** (*int*) – Length of send buffer.
- **root** (*int*) – the process who has the data to broadcast.

Returns The data sent from root process

Return type ys (numpy/cupy array)

abstract bcast_data(*model*)

Broadcast Chainer model parameter data

abstract bcast_obj(*obj*, *max_buf_len=None*, *root=0*)

Broadcasts an arbitrary object from root to all non-root processes.

Parameters

- **obj** – arbitrary object to broadcast to all other non-root processes. Will be ignored at all non-root processes.
- **max_buf_len** (*int*) – max length of the send buffer
- **root** (*int*) – rank of the root processes who sends an object

Returns an object sent from the root process.

finalize()
Finalizes and cleans up internal resource.

The communicator SHALL NOT be used after calling this `finalize()`. The behaviour is undefined when calling `finalize` on the same communicator multiple times.

abstract gather(*data*, *root=0*)
Gathers an ndarray from all processes to root process

Parameters

- **data** (`ndarray`, *or scaler*) – for root process this is ignored. For For non-root processes, the data to send to root process.
- **root** (*int*) – rank of the process who receives the data.

Returns For root process, the ndarray sent from non-root processes. For non-root processes, what?

abstract gather_obj(*obj*, *root=0*)
Gathers arbitrary objects from all non-root processes to the root.

Parameters

- **obj** – arbtrary object to send to root process. Root process will receive this argument included in returned list.
- **root** (*int*) – rank of the root node who receives all objects.

Returns A list of objects sent from all processes.

TODO(kuenishi): make sure the ordering of objects in the returned list.

get_config(*name=None*)
Get configuration value(s)

Parameters **name** (*str*) – Name of the configuration to get. If it is `None`, all config names and values are returned.

Returns Actual value of the configuration if it is on. `None` if it is off. If `None` is given as `name`, `None` or dictionary of names and configuration values is returned.

property inter_rank
The rank of this node in the cluster.

property inter_size
Number of nodes that participates the cluster.

property intra_rank
Intra rank (process id in the machine) of this process.

abstract multi_node_mean_grad(*model*, *zero_fill=False*)
mean Chainer model gradients.

Parameters

- **link** (`Link`) – Link object.
- **zero_fill** – A knob to control whether to fill gradients of initialized and unused Link (which is None internally) with zero-valued array, because the all gradients must be an array among processes for performing all-reduce, which might be an array or None after backward computation. Gradients of uninitialized Link are skipped. If it is False, gradients of unused Link are just skipped.

property rank
Rank (process id in the cluster) of this process in integer.

abstract recv(*source*, *tag*)
Receives an ndarray from source.

To receive the message, sender must send the data.

Parameters

- **source** (*int*) – Rank of the source process
- **tag** (*int*) – The tag to specifically receive the message

Returns The data sent from source process

abstract recv_obj(*source*, *tag*)
Receives an arbitrary Python object from source process with a tag.

Parameters

- **source** (*int*) – Rank number of sender process, to selectively receive the object.
- **tag** – tag to identify the message.

Returns an object sent from the source by `send_obj`.

abstract scatter(*xs*, *root=0*)
A primitive of inter-process scatter communication.

This method tries to invoke scatter communication within the communicator. All processes in the communicator are expected to invoke `scatter()`.

Parameters

- **xs** (*tuple of numpy/cupy array*) – Arrays to be scattered.
- **root** (*int*) – Rank of root process.

Returns Received arrays.

Return type ys (numpy/cupy array)

abstract send(*data*, *dest*, *tag*)
Sends an ndarray to destination

Receiver must invoke `recv()` to wait for the message.

Parameters

- **data** – data to be sent (tuple, list or raw numpy/cupy array)
- **dest** (*int*) – Rank of the destination process
- **tag** (*int*) – The tag to identify the message

abstract send_obj(*obj*, *dest*, *tag*)
Sends an arbitrary Python object to destination with a tag.

Parameters

- **obj** – Arbitrary object to send to receiver.
- **dest** (*int*) – Rank number of receiver process (destination).
- **tag** – tag to identify the message.

set_config(*name*, ***kwargs*)
Set configurations(s) on/off

The usage of configurations depends on each communicator. See `create_communicator()` for available configurations.

Parameters

- **name** (`str`) – Name of configuration to set.
- **value** – Give arbitrary object to set.
- **kwargs** – Arbitrary arguments depending on each configuration.

property size
Number of processes of the cluster.

abstract split(*color*, *key*)
A function anologous to `MPI_Comm_Split` .

This method splits the inter MPI commnicator and return a wrapped ChainerMN communicator.

Parameters

- **color** (`int`) – Index of new group. The process with the same color will be assigned to the same group.
- **key** (`int`) – Control of rank assignment. The process will be assigned a rank in the new group ordered by the value of key. If you do not care of the rank, you can just simply specify the original rank.

Returns CommunicatorBase

7.4.2 Optimizers and Evaluators

chainermn.**create_multi_node_optimizer**(*actual_optimizer*, *communicator*, *double_buffering=False*, *zero_fill=True*)
Create a multi node optimizer from a Chainer optimizer.

Parameters

- **actual_optimizer** – Chainer optimizer (e.g., `chainer.optimizers.Adam`).
- **communicator** – ChainerMN communicator.
- **double_buffering** – If `True`, all-reduce and other processing (such as forward and backward) are overlapped using double buffering. There are cases where accuracy is affected because the gradients of the previous iteration are used for update. This flag is supported by `PureNcclCommunicator` only.
- **zero_fill** – A knob to control whether to fill gradients of initialized and unused Link (which is None internally) with zero-valued array, because the all gradients must be an array among processes for performing all-reduce, which might be an array or None after backward computation. Gradients of uninitialized Link are skipped. If it is False, gradients of unused Link are just skipped.

Returns The multi node optimizer based on `actual_optimizer`.

chainermn.**create_multi_node_evaluator**(*actual_evaluator*, *communicator*)
Create a multi node evaluator from a normal evaluator.

Actually this method patches the evaluator to work in multi node environment. This method adds several hidden attributes starting with *_mn_* prefix.

Parameters

- **actual_evaluator** – evaluator to be patched (e.g., `chainer.training.extensions.Evaluator`)
- **communicator** – ChainerMN communicator

Returns The multi-node patched `actual_evaluator`.

Note: After patched, original evaluator does not work correctly in non-MPI environment.

---

class `chainermn.extensions.`**`GenericMultiNodeEvaluator`**(*comm*, *iterator*, *target*, *device=None*, *converter=<chainer.dataset.convert._ArbitraryCallableConverter object>*, *root=0*, ***kwargs*)

Generic multi-node evaluator for non-allreducable evaluation.

This is to evaluate a Dataset that cannot evenly divided across all processes in the communicator, for evaluation calculation that is not applicable to a simple add-and-devide style averaging among processes.

Users are recommeneded to implement its own local calculation `calc_local()` (e.g. at each distributed GPU) and aggregation `aggregate()` of its results. Although it has built-in implementaiton of those two methods.

It has several drawbacks; 1) Additional implementation of aggregation required to users, and 2) no compatibility with *`Evaluator`*.

Note: No automatic support of Reporter is provided; Set it up at `initialize()` method

---

Parameters

- **comm** – ChainerMN communicator object
- **iterator** – An iterator for test dataset. Must be non-repeated.
- **target** (*callable*) – A model to evaluate with test dataset
- **device** (*int or chainer.backend.Device*) – A device indicator to send data with converter. Not used when the converter is not using any devices.
- **converter** (*callable*) – A converter. Default value is *`chainer.dataset.concat_examples()`* .
- **root** (*int*) – Rank number of root process to run bcast and gather with.
- **progress_hook** (*callable*) – A callable that receives single argument for indicators. The callable is only callled at root process.

aggregate(*results*)

A generic aggregation method.

Override this method for original aggregation calculation. By default, it just does nothing but returns the input. This method is called once and only once accross the cluster, at root process. Reporting can be run here.

Parameters results (*list*) – List of return value of `calc_local()` obtained from all nodes..

calc_local(**args*, ***kwargs*)

A generic method for local calculation.

Override this method to run its local calculation. Otherwise, results are calculated with original target and test dataset.

Parameters

- **args** – Result of converter when it is tuple.
- **kwargs** – Result of converter when it is dict.

Returns Arbrary value may be returned, but must not be `None`.

7.4.3 Dataset Utilities

`chainermn.`**scatter_dataset**(*dataset*, *comm*, *root=0*, *shuffle=False*, *seed=None*, *max_buf_len=268435456*, *force_equal_length=True*)

Scatter the given dataset to the workers in the communicator.

The dataset of worker `root` (i.e., the worker whose `comm.rank` is `root`) is scattered to all workers. The given dataset of other workers are ignored. The dataset is split to sub datasets of almost equal sizes and scattered to workers. To create a sub dataset, `chainer.datasets.SubDataset` is used.

Note:: Make sure `force_equal_length` flag is *not* off for multinode evaluator or multinode updaters, which assume that the iterator has the same lengths among processes to work correctly.

Parameters

- **dataset** – A dataset (e.g., `list`, `numpy.ndarray`, `chainer.datasets.TupleDataset`, ...).
- **comm** – ChainerMN communicator or MPI4py communicator.
- **shuffle** (*bool*) – If `True`, the order of examples is shuffled before being scattered.
- **root** (*int*) – The root process of the scatter operation.
- **seed** (*int*) – Seed the generator used for the permutation of indexes. If an integer being convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the given dataset always belongs to a specific subset. If `None`, the permutation is changed randomly.
- **max_buf_len** (*int*) – Max buffer size to be used at broadcasting binaries. Must not be larger than 2147483647.
- **force_equal_length** (*bool*) – Force the scattered fragments of the dataset have equal length. If `True`, number of scattered examples is guaranteed to be equal among processes and scattered datasets may have duplication among processes. Otherwise, number of scattered examples may not be equal among processes, but scattered examples are guaranteed to have no duplication among processes, intended for strict evaluation of test dataset to avoid duplicated examples.

Returns Scattered dataset.

`chainermn.`**scatter_index**(*n_total_samples*, *comm*, *root=0*, *force_equal_length=True*)

Scatters only index to avoid heavy dataset broadcast

This is core functionality of `scatter_dataset`, which is almost equal to following code snippet:

```
(b, e) = scatter_index(len(dataset), comm)
order = None
if shuffle:
    order = numpy.random.RandomState(seed).permutation(
        n_total_samples)
    order = comm.bcast_obj(order)
dataset = SubDataset(dataset, b, e, order)
```

Note:: Make sure `force_equal_length` flag is *not* off for multinode evaluator or multinode updaters, which assume that the iterator has the same lengths among processes to work correctly.

Parameters

- **n_total_samples** (*int*) – number of total samples to scatter
- **comm** – ChainerMN communicator object
- **root** (*int*) – root rank to coordinate the operation
- **force_equal_length** (*bool*) – Force the scattered fragments of the index have equal length. If `True`, number of scattered indices is guaranteed to be equal among processes and scattered datasets may have duplication among processes. Otherwise, number of scattered indices may not be equal among processes, but scattered indices are guaranteed to have no duplication among processes, intended for strict evaluation of test dataset to avoid duplicated examples.

Returns Tuple of two integers, that stands for beginning and ending offsets of the assigned sub part of samples. The ending offset is not border inclusive.

`chainermn.datasets.`**`create_empty_dataset`**`(`*dataset*`)`

Creates an empty dataset for models with no inputs and outputs.

This function generates an empty dataset, i.e., `__getitem__()` only returns `None`. Its dataset is compatible with the original one. Such datasets used for models which do not take any inputs, neither return any outputs. We expect models, e.g., whose `forward()` is starting with `chainermn.functions.recv()` and ending with `chainermn.functions.send()`.

Parameters **dataset** – Dataset to convert.

Returns Dataset consists of only patterns in the original one.

Return type *TransformDataset*

7.4.4 Links

class `chainermn.`**`MultiNodeChainList`**`(`*comm*`)`

Combining multiple non-connected components of computational graph.

This class combines each `chainer.Chain`, which represents one of the non-connected component in compuational graph. In `__call__()`, the returned object of `chainer.Chain` (which represents pointer) are passed to the next `chainer.Chain`, in order to retain the computational graph connected and make backprop work properly.

Users add each `chainer.Chain` by `add_link()` method. Each chain is invoked in forward computation according to the order they are added, and in backward computation according to the reversed order.

Example (basic usage)

This is a simple example of the model which sends its outputs to rank=1 machine:

```
import chainer
import chainer.functions as F
import chainermn


class SimpleModelSub(chainer.Chain):

    def __init__(self, n_in, n_hidden, n_out):
        super(SimpleModelSub, self).__init__(
            l1=L.Linear(n_in, n_hidden),
            l2=L.Linear(n_hidden, n_out))

    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        return self.l2(h1)


class SimpleModel(chainermn.MultiNodeChainList):

    def __init__(self, comm, n_in, n_hidden, n_out):
        super(SimpleModel, self).__init__(comm)
        self.add_link(
            SimpleModelSub(n_in, n_hidden, n_out),
            rank_in=None,
            rank_out=1)
```

Example (split MLP on 2 processes)

This is the other example of two models interacting each other:

```
import chainer
import chainer.functions as F
import chainermn


class MLP(chainer.Chain):

    def __init__(self, n_in, n_hidden, n_out):
        super(MLP, self).__init__(
            l1=L.Linear(n_in, n_hidden),
            l2=L.Linear(n_hidden, n_hidden),
            l3=L.Linear(n_hidden, n_out))

    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)


class Model0(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(Model0, self).__init__(comm)
        self.add_link(
```

(continues on next page)

(continued from previous page)

```
            MLP(10000, 5000, 2000),
            rank_in=None,
            rank_out=1)
        self.add_link(
            MLP(100, 50, 10),
            rank_in=1,
            rank_out=None)


class Model1(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(Model1, self).__init__(comm)
        self.add_link(MLP(2000, 500, 100), rank_in=0, rank_out=0)
```

`Model0` is expected to be on rank=0, and `Model1` is expected to be on rank=1. The first `MLP` in `Model0` will send its outputs to `Model1`, then `MLP` in `Model1` will receive it and send its outputs to the second `MLP` in `Model0`.

Example (sending tuples)

This is the example for sending a tuple:

```
import chainer
import chainer.functions as F
import chainermn

class NN0(chainer.Chain):
    def __call__(self, x):
        y0 = some_calculation_nn0_0(x)
        y1 = some_calculation_nn1_1(x)
        return y0, y1

class NN1(chainer.Chain):
    def __call__(self, y):
        y0, y1 = y  # unpack tuple from NN0
        return some_calculation_nn1(y0, y1)

class Model_on_Process_0(chainermn.MultiNodeChainList):
    def __init__(self, comm):
        super(Model_on_Process_0, self).__init__(comm=comm)
        self.add_link(NN0(), rank_in=None, rank_out=1)

class Model_on_Process_1(chainermn.MultiNodeChainList):
    def __init__(self, comm):
        super(Model_on_Process_1, self).__init__(comm=comm)
        self.add_link(NN1(), rank_in=0, rank_out=None)
```

In this example, `Model_on_Process_0` sends two elemental tuple `(y0, y1)` (returned by `NN0.__call__`) to `Model_on_Process_1`, which can be unpacked as shown in `NN1.__call__`.

Parameters **comm** (*chainermn.communicators._base.CommunicatorBase*) – ChainerMN communicator.

add_link(*link*, *rank_in=None*, *rank_out=None*)
Register one connected link with its inout rank.

Parameters

- **link** (chainer.Link) – The link object to be registered.
- **rank_in** (*int, list, or None*) – Ranks from which it receives data. If None is specified, the model does not receive from any machines.
- **rank_out** (*int, list, or None*) – Ranks to which it sends data. If None is specified, the model will not send to any machine.

class chainermn.links.**MultiNodeBatchNormalization**(*size*, *comm*, *decay=0.9*, *eps=2e-05*, *dtype=None*, *use_gamma=True*, *use_beta=True*, *initial_gamma=None*, *initial_beta=None*, *communication_backend='auto'*)
Batch normalization layer that can use the whole batch stats.

When using chainer.link.BatchNormalization, batch mean and std are computed independently for the local batch in each worker. When local batch size is too small, training is unstable due to unreliable batch stats.

In contrast, when using this MultiNodeBatchNormalization, workers communicate to conduct 'correct' batch normalization (e.g., obtaining mean and std for the whole global batch).

This link works only with Chainer >= 2.0.0.

Parameters

- **size** (*int or tuple of ints*) – Size (or shape) of channel dimensions.
- **comm** (*ChainerMN communicator*) – communicator to share the batch stats.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability.
- **dtype** (*numpy.dtype*) – Type to use in computing.
- **use_gamma** (*bool*) – If `True`, use scaling parameter. Otherwise, use unit(1) which makes no effect.
- **use_beta** (*bool*) – If `True`, use shifting parameter. Otherwise, use unit(0) which makes no effect.
- **communication_backend** (*str*) – `mpi`, `nccl` or `auto`. It is used to determine communication backend. If `auto`, use the best communication backend for each communicator.

chainermn.links.**create_mnbn_model**(*link*, *comm*, *communication_backend='auto'*)
Create a link object with MultiNodeBatchNormalization.

Returns a copy of *link*, where BatchNormalization is replaced by MultiNodeBatchNormalization.

Parameters

- **link** – Link object
- **comm** – ChainerMN communicator
- **communication_backend** (*str*) – `mpi`, `nccl` or `auto`. It is used to determine communication backend of MultiNodeBatchNormalization. If `auto`, use the best communication backend for each communicator.

Returns Link object where BatchNormalization is replaced by MultiNodeBatchNormalization.

7.4.5 Functions

chainermn.functions.**send**(*x*, *communicator*, *rank*, *tag=0*)

Send elements to target process.

This function returns a dummy variable only holding the computational graph. If `backward()` is invoked by this dummy variable, it will try to receive gradients from the target process and send them back to the parent nodes.

Parameters

- **x** (Variable) – Variable holding a matrix which you would like to send.
- **communicator** (*chainer.communicators.CommunicatorBase*) – ChainerMN communicator.
- **rank** (*int*) – Target process specifier.
- **tag** (*int*) – Optional message ID (MPI feature).

Returns A dummy variable with no actual data, only holding the computational graph. Please refer `chainermn.functions.pseudo_connect` for detail.

Return type *Variable*

chainermn.functions.**recv**(*communicator*, *rank*, *delegate_variable=None*, *tag=0*, *force_tuple=False*)

Receive elements from target process.

This function returns data received from target process. If `backward()` is invoked, it will try to send gradients to the target process. The received array will be on the current CUDA device if the corresponding `send()` is invoked with arrays on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Note: If you define non-connected computational graph on one process, you have to use `delegate_variable` to specify the output of previous computational graph component. Otherwise `backward()` does not work well. Please refer `chainermn.functions.pseudo_connect` for detail.

Parameters

- **communicator** (*chainer.communicators.CommunicatorBase*) – ChainerMN communicator.
- **rank** (*int*) – Target process specifier.
- **delegate_variable** (chainer.Variable) – Pointer to the other non-connected component.
- **tag** (*int*) – Optional message ID (MPI feature).
- **force_tuple** (*bool*) – If `False` (the default) a Variable will be returned when the number of outputs is one. Otherwise, this method returns a tuple even when the number of outputs is one.

Returns Data received from target process. If `backward()` is invoked by this variable, it will send gradients to the target process.

Return type *Variable*

`chainermn.functions.pseudo_connect` (*delegate_variable*, **actual_variables*)

Connect independent connected graph component.

This function is implemented to return received arguments directly, except the first `delegate_variable`. In backward computation, it returns received gradients directly, adding a zero grad corresponding to `delegate_variable`. The detail of `delegate_variable` is described in the following notes.

Note: In model-parallel framework, models on each process might have many non-connected components. Here we call a given graph non-connected when multiple inter-process communications are needed for its computation. For example, consider the following example:

```
class ConnectedGraph(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(ConnectedGraph, self).__init__(comm)
        self.add_link(ConnectedGraphSub(), rank_in=3, rank_out=1)
```

This model receives inputs from rank=3 process and sends its outputs to rank=1 process. The entire graph can be seen as one connected component `ConnectedGraphSub`. Please refer the documentation of `MultiNodeChainList` for detail.

On the other hand, see the next example:

```
class NonConnectedGraph(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(NonConnectedGraph, self).__init__(comm)
        self.add_link(NonConnectedGraphSubA(), rank_in=3, rank_out=1)
        self.add_link(NonConnectedGraphSubB(), rank_in=1, rank_out=2)
```

This model consists of two components: at first, `NonConnectedGraphSubA` receives inputs from rank=3 process and sends its outputs to rank=1 process, and then `NonConnectedGraphSubB` receives inputs from rank=1 process and sends its outputs to rank=2 process. Here multiple inter-process communications are invoked between `NonConnectedGraphSubA` and `NonConnectedGraphSubB`, so it is regarded as non-connected.

Such kind of non-connected models can be problematic in backward computation. Chainer traces back the computational graph from the output variable, however naive implementation of `chainermn.functions.recv` does not take any inputs rather receives inputs by `MPI_Recv`, where backward path vanishes.

To prevent this, dummy variables what we call `delegate_variable` are used. In principle, `chainermn.functions.send` does not return any outputs because it sends data to the other process by `MPI_Send`. However, `chainermn.functions.send` returns a dummy / empty variable in our implementation, which is called `delegate_variable`. This variable does not hold any data, just used for retaining backward computation path. We can guarantee the backward computation just by putting `delegate_variable` to the next `chainermn.functions.recv` (`chainermn.functions.recv` has an optional argument to receive `delegate_variable`).

Note: In some cases the intermediate graph component returns model outputs. See the next example:

```
class NonConnectedGraph2(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(NonConnectedGraph2, self).__init__(comm)
        self.add_link(NonConnectedGraphSubA(), rank_in=1, rank_out=None)
        self.add_link(NonConnectedGraphSubB(), rank_in=None, rank_out=1)
```

This model first receives inputs from rank=1 process and make model outputs (specified by `rank_out=None`) in `NonConnectedGraphSubA`. Then using model inputs (specified by `rank_in=None`), `NonConnectedGraphSubB` sends its outputs to rank=1 process. Since `MultiNodeChainList.__call__` returns outputs of the last component (in this case, outputs of `NonConnectedGraphSubB`), naive implementation cannot output the returned value of `NonConnectedGraphSubA` as the model outputs. In this case, `pseudo_connect` should be used.

`pseudo_connect` takes two arguments. The first one `delegate_variable` is what we explained in above note. In this case, returned value of `NonConnectedGraphSubB` corresponds to `delegate_variable`. The second one `actual_variables` is "what we want `delegate_variable` to imitate". In `NonConnectedGraph2`, we obtain returned value of `NonConnectedGraphSubB` as the model outputs, but what we actually want is returned value of `NonConnectedGraphSubA`. At the same time we want to trace back this resulted variable in backward computation. Using `pseudo_connect`, we can make a variable whose data is the same as the returned value of `NonConnectedGraphSubA`, and which traces back `NonConnectedGraphSubB` first.

`pseudo_connect` should also be used in some pathological cases, for example, where multiple `chainermn.functions.send` occurs sequentially.

Parameters

- **delegate_variable** (`chainer.Variable`) – Pointer to the previous non-connected graph component.
- **actual_variables** (*tuple of chainer.Variable*) – Actual values which `delegate_variable` imitate.

Returns A variable with the given values combined with delegating variable.

Return type tuple of chainer.Variable

`chainermn.functions.`**`bcast`**(*comm, x, root=0*)

Differentiable broadcast communication between workers.

This function invokes broadcast communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are gathered to the root process and summed up.

The received array will be on the current CUDA device if `x` on the invoking process is on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Parameters

- **comm** – ChainerMN communicator.
- **x** (`chainer.Variable`) – Variable to be sent.

Returns Broadcasted variable.

Return type y (*chainer.Variable*)

`chainermn.functions.`**`gather`**(*comm, x, root=0*)

Differentiable gather communication between workers.

This function invokes gather communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are scattered from the root process to each slave.

The received array will be on the current CUDA device if `x` on the root process is on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Parameters

- **comm** – ChainerMN communicator.
- **x** (`chainer.Variable`) – Variable to be sent.

Returns Gathered variables. `None` for slaves.

Return type ys (*chainer.Variable*)

`chainermn.functions.`**`scatter`**(*comm*, *xs*, *root=0*)

Differentiable scatter communication between workers.

This function invokes scatter communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are gathered to the root process.

The received array will be on the current CUDA device if `xs` on the root process is on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Parameters

- **comm** – ChainerMN communicator.
- **xs** (*`list of chainer.Variable`*) – Variables to be scattered for master process. `None` for slave process.

Returns Scattered variable.

Return type y (*chainer.Variable*)

`chainermn.functions.`**`alltoall`**(*comm*, *xs*)

Differentiable all-to-all communication between workers.

This function invokes all-to-all communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, just passing input gradients back. Unlike point-to-point communication such as `chainermn.functions.send` and `chainermn.functions.recv`, users need not to care about delegate variables, since `backward()` will not be invoked until all gradients from output direction arrive. Please refer to `chainermn.functions.pseudo_connect` about the detail of delegate variables.

The received array will be on the current CUDA device on the invoking process if `xs` is on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Parameters

- **comm** – ChainerMN communicator.
- **xs** (*`list of chainer.Variables`*) – Variables to send.

Returns Received variables.

Return type ys (list of chainer.Variables)

`chainermn.functions.`**`allgather`**(*comm*, *x*)

Differentiable all-gather communication between workers.

This function invokes gather communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are reduced to each process.

The received array will be on the current CUDA device on the invoking process if `x` is on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Parameters

- **comm** – ChainerMN communicator.
- **x** (*chainer.Variables*) – Variables to send.

Returns Received variables.

Return type ys (list of chainer.Variables)

7.4.6 Iterators

`chainermn.iterators.`**`create_multi_node_iterator`**(*actual_iterator*, *communicator*, *rank_master=0*)

Create a multi node iterator from a Chainer iterator.

This iterator shares the same batches on multiple processes, simply broadcasting batches from master process to slave processes in each iteration. Master process obtains batches from `actual_iterator`, which you can specify any Chainer iterator (e.g. `chainer.iterators.SerialIterator`).

Here is an example situation. When we train a sequence-to-sequence model, where the encoder and the decoder is located on two different processes, we want to share the same batches on each process, thus inputs for the encoder and output teacher signals for the decoder become consistent.

In order to use the multi node iterator, first create the iterator from Chainer iterator and ChainerMN communicator:

```
iterator = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(
        dataset, batch_size, shuffle=True),
    communicator)
```

Then you can use it as the ordinary Chainer iterator:

```
updater = chainer.training.StandardUpdater(iterator, optimizer)
trainer = training.Trainer(updater)
trainer.run()
```

Since this iterator shares batches through network in each iteration, communication might be large. If you train your model-parallel network on extremely large dataset, you can also consider to use `chainermn.iterators.create_synchronized_iterator`.

Current multi node iterator supports numpy.float32 or tuple of numpy.float32 as the data type of the batch element.

Note: `create_multi_node_iterator` and `serialize` of created iterators must be called at the same time by master and slaves, unless it falls into deadlock because they synchronize internal states of iterators.

Parameters

- **actual_iterator** – Chainer iterator (`chainer.iterators.SerialIterator` and `chainer.iterators.MultiprocessIterator` are supported).
- **communicator** – ChainerMN communicator.

- **rank_master** – process rank to be master.

Returns The master-slave iterator based on `actual_iterator`.

`chainermn.iterators.`**`create_synchronized_iterator`**(*actual_iterator*, *communicator*)

Create a synchronized iterator from a Chainer iterator.

This iterator shares the same batches on multiple processes, using the same random number generators to maintain the order of batch shuffling same.

Here is an example situation. When we train a sequence-to-sequence model, where the encoder and the decoder is located on two different processes, we want to share the same batches on each process, thus inputs for the encoder and output teacher signals for the decoder become consistent.

In order to use the synchronized iterator, first create the iterator from Chainer iterator and ChainerMN communicator:

```
iterator = chainermn.iterators.create_synchronized_iterator(
    chainer.iterators.SerialIterator(
        dataset, batch_size, shuffle=True),
    communicator)
```

Then you can use it as the ordinary Chainer iterator:

```
updater = chainer.training.StandardUpdater(iterator, optimizer)
trainer = training.Trainer(updater)
trainer.run()
```

The resulting iterator shares the same shuffling order among processes in the specified communicator.

Parameters

- **actual_iterator** – Chainer iterator (e.g., `chainer.iterators.SerialIterator`).
- **communicator** – ChainerMN communicator.

Returns The synchronized iterator based on `actual_iterator`.

7.4.7 Trainer extensions

class `chainermn.extensions.`**`AllreducePersistent`**(*model*, *comm*)

Chainer extension to averagize persistents over workers.

When called, this extension invokes all-reduce communication among workers to compute averages of persistent variables in the model. Persistent variables are updated to the averages. Currently, we ignore integer persistent variables, and only float persistent variables are handled.

This extension is mainly to improve the running mean and variance of BatchNormalization by increasing the effective number of examples. We do not need to call this frequently; call just before storing or evaluating the model.

Parameters

- **model** (*chainer.link.Link*) – Target link object.
- **comm** (*ChainerMN communicator*) – communicator to compute averages.

`chainermn.extensions.`**`multi_node_snapshot`**(*comm*, *snapshot*, *replica_sets*)

Create trainer extension for multi-node snapshots

Provides generis multi-node snapshot saving and auto-load feature at multi-node environment, leveraging power of single-node snapshot.

In many cases snapshot target may differ, e.g. only trainer of rank 0 process often has extensions such as `LogReport` and so on, to not confuse terminal output. Just loading at one process and broadcasting it to other processes does not work in that case.

This wrapper addresses that issue by defining sets of replicas where within the set the target object is replicated and supposed to be same among processes. For example, a trainer example, only the trainer at rank `0` has special extensions and others doesn't:

```
trainer = Trainer(updater)
if comm.rank == 0:
    trainer.extend(extensions.DumpGraph('main/loss'))
    trainer.extend(extensions.LogReport())
    trainer.extend(extensions.PrintReport(
        ['epoch', 'main/loss', 'validation/main/loss',
         'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))
    trainer.extend(extensions.ProgressBar())
```

This case can be described with two replica sets, where each set can be represented as single integer that indicates rank number, or iterable set/list/generator of integers like this:

```
replica_sets = [[0], range(1, comm.size)]
```

Here the first replica set is described as `[0]`, or simply in short just `0`, and the second replica set is `range(1, comm.size)`, representing rest of processes other than `0`. The remaining list can be ommited. Thus in that case, it can be simplified more:

```
replica_sets = [0,]
```

In this case, the snapshot will be saved at rank `0` process and at rank `1` process. The latter represents the replica set of `range(1, comm.size)` . In this case autoloading at initialization of snapshot extension works after the restart cleanly, even though the size of the communicator differs.

Once the replica sets are defined, it can be easily extended:

```
replica_sets = [0,]
snapshot = multi_node_snapshot(comm, extensions.snapshot(),
                               replica_sets)
trainer.extend(snapshot, trigger=(1, 'epoch'))
```

More example tuples of replica set representation follows:

code	nproc	actual sets
`[0]`	4	`[{0}, {1, 2, 3}]`
`[0, 1]`	4	`[{0}, {1}, {2, 3}]`
`[0, 1], [2, 3]]`	4	`[{0, 1}, {2, 3}]`
`[]`	4	`[{0, 1, 2, 3}]`
`[range(0, 8, 2)]`	8	`[set(range(0, 8, 2)), set(range(1, 8, 2))]`

Parameters

- **comm** (*ChainerMN communicator*) – communicater object
- **snapshot** – Snapshot extension object obtained via *snapshot()* .

- **replica_sets** – list of replica set definition, where a replica set can be defined by single integer as rank number, or iterable integers.

Returns Trainer extension that wraps `snapshot` and properly controles number of snapshots.

chainermn.**create_multi_node_checkpointer**(*name*, *comm*, *cp_interval=5*, *gc_interval=5*, *path=None*)

Create multi-node checkpointer object

Generational snapshot extension to allow fault tolerance; It keeps several old snapshots to rollback synchronized snapshot at each MPI process. Snapshot files are identified as '<name>.<rank>.<iteration>'.

- <name> … identifier of the run where snapshot is kept for
- <rank> … which process owned the model
- <iteration> … number of iteration.

This extension keeps several files for each execution and allows users to resume the whole job at the latest snapshots of each MPI process, and the iteration where all snapshots agrees.

As this object is a usual Chainer extension, users can just create this object and pass to the trainer as an extension:

```
checkpointer = create_multi_node_checkpointer(name=run_id, comm=comm)
trainer.extend(checkpointer, trigger=(25, 'iteration'))
```

To run recovery at startup, before first iteration, run

checkpointer.maybe_load(trainer, optimizer)

before `trainer.run()` . If nothing is recovered (i.e. no snapshot found), `trainer.updater.iteration` will remain `0` . Otherwise it will have the value of snapshot and the training will resume from that iteration. `optimizer` is optional but this will let multi node optimizer avoid initial broadcast when all snapshot data among nodes are all in sync.

Note: Make sure that `checkpointer.maybe_load` is called *after* all extensions with states, such as `ExponentialShift`, set to the trainer.

After training finished without errors all those temporary checkpoints will be cleaned up at all nodes.

Another example to use checkpointer *without* trainer would be:

```
checkpointer = create_multi_node_checkpointer(name=run_id, comm=comm)
checkpointer.maybe_load(obj_you_want_to_snap, optimizer)

while True: ## Training loop
    ...
    updater.update()
    ...
    checkpointer.save(obj_you_want_to_snap)  # Make a checkpoint
```

Parameters

- **name** (*str*) – unique id of the run
- **comm** – communicater in ChainerMN
- **cp_interval** (*int*) – minimum number of checkpoints to preserve
- **gc_interval** (*int*) – interval to collect non-preserved checkpoints

7.4.8 Configurations

Environmental Variables

CHAINERMN_FORCE_ABORT_ON_EXCEPTIONS If this variable is set to a non-empty value, ChainerMN installs a global hook to Python's *sys.excepthook* to call `MPI_Abort()` when an unhandled exception occurs. See *MPI process hangs after an unhandled Python exception.*

ChainerMN issue #236 may also help to understand the problem.

Execution Control

`chainermn.global_except_hook.`**`add_hook`**`()`

Add a global hook function that captures all unhandled exceptions.

The function calls MPI_Abort() to force all processes abort. It is useful when you run your training script on a cloud platform.

CHAPTER

EIGHT

EXPORT CHAINER TO ONNX

8.1 Introduction

ONNX-Chainer converts Chainer model to ONNX format, export it.

8.1.1 Installation

Install dependencies using `pip` via PyPI:

```
$ pip install packaging 'onnx<1.6.1'
```

8.1.2 Quick Start

First, install ChainerCV to get the pre-trained models.

```
import numpy as np

import chainer
import chainercv.links as C
import onnx_chainer

model = C.VGG16(pretrained_model='imagenet')

# Pseudo input
x = np.zeros((1, 3, 224, 224), dtype=np.float32)

onnx_chainer.export(model, x, filename='vgg16.onnx')
```

`vgg16.onnx` file will be exported.

8.1.3 Supported Functions

Currently 82 Chainer Functions are supported to export in ONNX format.

Activation

- ClippedReLU
- ELU
- HardSigmoid

- LeakyReLU
- LogSoftmax
- PReLUFunction
- ReLU
- Sigmoid
- Softmax
- Softplus
- Tanh

Array

- Cast
- Concat
- Copy
- Depth2Space
- Dstack
- ExpandDims
- GetItem
- Hstack
- Pad[1 2]
- Repeat
- Reshape
- ResizeImages
- Separate
- Shape[5]
- Space2Depth
- SplitAxis
- Squeeze
- Stack
- Swapaxes
- Tile
- Transpose
- Vstack
- Where

Connection

- Convolution2DFunction

[1] mode should be either 'constant', 'reflect', or 'edge'
[2] ONNX doesn't support multiple constant values for Pad operation
[5] Chainer doesn't support Shape function

- ConvolutionND
- Deconvolution2DFunction
- DeconvolutionND
- EmbedIDFunction[3]
- LinearFunction

Loss

- SoftmaxCrossEntropy

Math

- Absolute
- Add
- AddConstant
- ArgMax
- ArgMin
- BroadcastTo
- Clip
- Div
- DivFromConstant
- Exp
- Identity
- LinearInterpolate
- LogSumExp
- MatMul
- Max
- Maximum
- Mean
- Min
- Minimum
- Mul
- MulConstant
- Neg
- PowConstVar
- PowVarConst
- PowVarVar
- Prod
- RsqrtGPU

[3] Current ONNX doesn't support ignore_label for EmbedID

- Sqrt
- Square
- Sub
- SubFromConstant
- Sum

Noise

- Dropout[4]

Normalization

- BatchNormalization
- FixedBatchNormalization
- LocalResponseNormalization
- NormalizeL2

Pooling

- AveragePooling2D
- AveragePoolingND
- MaxPooling2D
- MaxPoolingND
- ROIPooling2D
- Unpooling2D

8.1.4 Tested Environments

- OS
  - Ubuntu 16.04, 18.04
  - Windows 10
- Python 3.5.5, 3.6.7, 3.7.2
- ONNX 1.4.1, 1.5.0
  - opset version 7, 8, 9, 10
- Chainer stable, preview
- ONNX-Runtime 0.5.0

8.1.5 Run Test

1. Install test modules

First, test modules for testing:

[4] In test mode, all dropout layers aren't included in the exported file

```
$ pip install -e .[test]
$ pip install onnxruntime
```

Test on GPU environment requires Cupy:

```
$ pip install cupy  # or cupy-cudaXX is useful
```

2. Run tests

Next, run `pytest`:

```
$ pytest -m "not gpu" tests/onnx_chainer_tests
```

on GPU environment:

```
$ pytest tests/onnx_chainer_tests
```

8.1.6 Contribution

Any contribution to ONNX-Chainer is welcom!

- Python codes follow Chainer Coding Guidelines

8.2 Module Reference

8.2.1 Export

ONNX-Chainer exports Chainer model to ONNX graph with various options.

`onnx_chainer.export`
`onnx_chainer.export_testcase`

8.2.2 Export Utilities

ONNX-Chainer provides some utility functions to help exporting.

`onnx_chainer.replace_func.fake_as_funcnode`
`onnx_chainer.replace_func.as_funcnode`

8.2.3 Convert Utilities

These utilities helps converting from Chainer model to ONNX format, mainly used them internally.

`onnx_chainer.context.Context`

onnx_chainer.onnx_helper.GraphBuilder
onnx_chainer.onnx_helper.set_func_name
onnx_chainer.onnx_helper.get_func_name
onnx_chainer.onnx_helper.make_node
onnx_chainer.onnx_helper.write_tensor_pb
onnx_chainer.onnx_helper.cleanse_param_name

8.2.4 Testing Utilities

onnx_chainer.testing.input_generator.increasing
onnx_chainer.testing.input_generator.nonzero_increasing
onnx_chainer.testing.input_generator.positive_increasing

8.3 Indices and tables

- genindex
- search

CHAPTER

NINE

API COMPATIBILITY POLICY

This documentation explains the design policy on compatibilities of Chainer APIs. Development team should follow this policy on deciding to add, extend, and change APIs and their behaviors.

This documentation is written for both users and developers. Users can decide the level of dependencies on Chainer's implementations in their codes based on this document. Developers should read through this documentation before creating pull requests that contain changes on the interface. Note that this documentation may contain ambiguities on the level of supported compatibilities.

9.1 Versioning and Backward Compatibility

The versioning of Chainer follows the PEP 440 and a part of Semantic versioning. See *Contribution Guide* for details of versioning.

The backward compatibility is kept for **revision updates** and **minor updates**, which are applied to the stable version. A **major update** from the latest release candidate basically keeps the backward compatibility, although it is not guaranteed. Any **pre-releases** may break the backward compatibility.

9.2 Breaking the Compatibility

We sometimes need to break the backward compatibility to improve the framework design and to support new kinds of machine learning methods. Such a change is only made into pre-releases (alpha, beta, and release candidate) and sometimes into the major update.

A change that breaks the compatibility affects user codes. We try to lower the cost of adapting your code to the newer version. The following list shows an example of what we can do to reduce the cost (*Note: this is not a promise; what kind of actions we can take depends on the situation*).

- When an argument is removed from an existing API, passing the argument to the updated API will emit an error with a special error message. The error message tells you how to fix your code.
- When a function or a class is removed, we make the current stable version emit a deprecation warning. **Note that the deprecation warning is not printed by default in Python.** You have to manually turn on the deprecation warning by `warnings.simplefilter('always', DeprecationWarning)`.
- When a definition of a link is changed, we try to enable it to deserialize a model dumped with an older version of Chainer. In most cases, we cannot guarantee that a model serialized with a newer version of Chainer is loadable by an older version of Chainer.

9.3 Experimental APIs

Thanks to many contributors, we have introduced many new features to Chainer.

However, we have sometimes released new features only to later notice that their APIs are not appropriate. In particular, we sometimes know that the API is likely to be modified in the near future because we do not have enough knowledge about how well the current design fits to the real usages. **The objective of experimental APIs is to declare that the APIs are likely to be updated in the near future so that users can decide if they can(not) use them.**

Any newly added API can be marked as *experimental*. Any API that is not experimental is called *stable* in this document.

Note: Undocumented behaviors are not considered as APIs, so they can be changed at any time (even in a revision update). The treatment of undocumented behaviors are described in *Undocumented behaviors* section.

When users use experimental APIs for the first time, warnings are raised once for each experimental API, unless users explicitly disable the emission of the warnings in advance.

See the documentation of `chainer.utils.experimental()` to know how developers mark APIs as experimental and how users enable or disable the warnings practically.

Note: It is up to developers if APIs should be annotated as experimental or not. We recommend to make the APIs experimental if they implement large modules or make a decision from several design choices.

9.4 Supported Backward Compatibility

This section defines backward compatibilities that revision updates must maintain.

9.4.1 Documented Interface

Chainer has the official API documentation. Many applications can be written based on the documented features. We support backward compatibilities of documented features. In other words, codes only based on the documented features run correctly with revision-updated versions.

Developers are encouraged to use apparent names for objects of implementation details. For example, attributes outside of the documented APIs should have one or more underscores at the prefix of their names.

Note: Although it is not stated as a rule, we also try to keep the compatibility for any interface that looks like a stable feature. For example, if the name of a symbol (function, class, method, attribute, etc.) is not prefixed by an underscore and the API is not experimental, the API should be kept over revision updates even if it is not documented.

9.4.2 Undocumented behaviors

Behaviors of Chainer implementation not stated in the documentation are undefined. Undocumented behaviors are not guaranteed to be stable between different revision versions.

Even revision updates may contain changes to undefined behaviors. One of the typical examples is a bug fix. Another example is an improvement on implementation, which may change the internal object structures not shown in the documentation. As a consequence, **even revision updates do not support compatibility of pickling, unless the full layout of pickled objects is clearly documented.**

9.4.3 Documentation Error

Compatibility is basically determined based on the documentation, although it sometimes contains errors. It may make the APIs confusing to assume the documentation always stronger than the implementations. We therefore may fix the documentation errors in any updates that may break the compatibility in regard to the documentation.

Note: Developers should not fix the documentation and implementation of the same functionality at the same time in revision updates as a "bug fix" unless the bug is so critical that no users are expected to be using the old version correctly.

9.4.4 Object Attributes and Properties

Object attributes and properties are sometimes replaced by each other. It does not break the user codes, except the codes depend on how the attributes and properties are implemented.

9.4.5 Functions and Methods

Methods may be replaced by callable attributes keeping the compatibility of parameters and return values. It does not break the user codes, except the codes depend on how the methods and callable attributes are implemented.

9.4.6 Exceptions and Warnings

The specifications of raising exceptions are considered as a part of standard backward compatibilities. No exception is raised in the future revision versions with correct usages that the documentation allows.

On the other hand, warnings may be added at any revision updates for any APIs. It means revision updates do not keep backward compatibility of warnings.

9.5 Model Format Compatibility

Links and chains serialized by official serializers that Chainer provides are correctly loaded with the future versions. They might not be correctly loaded with Chainer of the lower versions.

Note: Current serialization APIs do not support versioning. It prevents us from introducing changes in the layout of objects that support serialization. We are discussing versioning in serialization APIs.

9.6 Installation Compatibility

The installation process is another concern of compatibilities.

Any changes on the set of dependent libraries that force modifications on the existing environments should be done in pre-releases and major updates. Such changes include following cases:

- dropping supported versions of dependent libraries (e.g. dropping cuDNN v2)
- adding new mandatory dependencies (e.g. adding h5py to setup_requires)

Note: We sometimes have to narrow the supported versions due to bugs in the specific versions of libraries. In such a case, we may drop the support of those versions even in revision updates unless a workaround is found for the issue.

CHAPTER

TEN

CONTRIBUTION GUIDE

Chainer is an open source software hosted on GitHub and welcomes contributors to take part in the development of the framework. This is a document aimed towards such contributors. Anyone who for instance would like to file an issue or send a pull request (PR) is encouraged to go through it.

10.1 Issues and Pull Requests

First steps in contributing to Chainer often involve filing an issue or creating a PR. This section describes how to do so.

10.1.1 How to File an Issue

To file an issue on GitHub, you often only need to follow instructions given by the template. Write precise explanations on how you want Chainer to behave or include necessary and sufficient conditions to reproduce the bugs. Feature requests should include **what** you want to do and preferably **why**. You may additionally suggest **how**.

> **Warning:** If you have a question regarding the usage of Chainer, it is recommended that you send a post to StackOverflow or the Chainer User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices.

10.1.2 How to Send a Pull Request

If you can write code to fix an issue, it is encouraged to send a PR.

In that case, confirm the following points before starting to write any code.

- Read *Coding Guidelines* and *Unit Testing*.
- Check the appropriate branch to which you should send a PR, following *Git Branches*. If you are unsure about which branch to target, choose the `master` branch. The current source tree of the chosen branch is the starting point of your change.

After writing your code **(including unit tests and hopefully documentations!)**, send a PR on GitHub. You have to write a precise explanation of **what** and **how** in the description; this is the first documentation of your code and an important part of your PR.

However, even if your code is not complete, you can send a PR as a *work-in-progress (WIP) PR* by prefixing the PR title with `[WIP]`. If you just describe the PR, the core team and other contributors can join the discussion about how to proceed with it. WIP PRs may occasionally be useful for discussing based on concrete code.

When a PR is created (or updated), it is automatically tested in one of our CI environments, namely Travis CI. There are other CI environments as well often manually triggered by the reviewer. The various CIs are required to test for instance different platforms or CUDA environments. Once the tests in all CI environments pass and/or the PR is approved by the reviewer, the PR will be merged.

Note: If you are planning to add a new feature or modify existing APIs, **it is recommended that you open an issue and discuss the design first.** Following the consequences of the discussions, you can send a PR that is smoothly reviewed in a shorter time.

10.1.3 Issue/Pull Request Labels

Issues and PRs are labeled on GitHub so that they can be grouped, filtered and better maintained. For instance, a label can indicate that a ticket needs response from the PR author, or that an issue needs immediate action in case of a critical bug. Please refer to the list of lables on GitHub.

10.2 Coding Guidelines

We follow PEP 8 and partially OpenStack Style Guidelines as basic style guidelines. Any contributions in terms of code are expected to follow these guidelines.

You can use the `autopep8` and the `flake8` commands to check whether or not your code follows the guidelines. In order to avoid confusion from using different tool versions, we pin the versions of those tools. Install them with the following command (from within the top directory of the Chainer repository):

```
$ pip install -e '.[stylecheck]'
```

And check your code with:

```
$ autopep8 path/to/your/code.py
$ flake8 path/to/your/code.py
```

`autopep8` can automatically correct Python code to conform to the PEP 8 style guide:

```
$ autopep8 --in-place path/to/your/code.py
```

The `flake8` command lets you know parts of your code that are not following the style guidelines.

Note that `flake8` command is not perfect. It does not check some of the style guidelines. Here is a (not-exhaustive) list of the rules that `flake8` cannot check.

- Relative imports are prohibited. [H304]
- Importing non-module symbols is prohibited.
- Import statements must be organized into three parts: standard libraries, third-party libraries, and internal imports. [H306]

In addition, we restrict the usage of *shortcut aliases* in any global-scope code. In particular, you cannot use shortcut aliases to designate a parent class in global-scope class definitions. When you want to make a class inheriting another class defined in another module, you have to spell out the full module name instead of importing a module that provides an alias.

For example, the following code is not allowed.

```
import chainer

class MyLink(chainer.Link): ...
```

Instead, import `chainer.link` and use that.

```
import chainer.link

class MyLink(chainer.link.Link): ...
```

If you feel the code too verbose, you can also use `from import` or `import as`.

```
from chainer import link

class MyLink(link.Link): ...
```

Note: From v3.0, we allow shortcut aliases used inside of functions and methods that are not called from any global scope code. For example, you can write `chainer.Variable` instead of `chainer.variable.Variable` inside of functions and methods. Use of such aliases was prohibited in the past for avoiding confusing errors related to cyclic dependencies; we relaxed the rule so that the library code looks similar to user code.

When you use such shortcut aliases, please be careful of cyclic imports. One of the typical pitfalls is a way to import `chainer.functions`. An import like `import chainer.functions as F` within modules under `chainer.functions` does not work. An import like `from chainer import functions` works well with Python 3, but does not with Python 2. We recommend that you use `import chainer.functions` and spell out like `chainer.functions.foo` in your methods.

10.3 Unit Testing

Testing is one of the most important aspects of your PR. You should write test cases and verify your implementation by following the testing guide above. If you modify code related to existing unit tests, you must run appropriate commands and confirm that the tests still pass.

Note that we are using `pytest` and the `mock` package for testing. They are not included in Chainer and need to be installed as follows:

```
$ pip install pytest mock
```

10.3.1 How to Run Tests

You can run all unit tests with the following command from the root directory of the Chainer:

```
$ python -m pytest
```

Or specify a test script that you want to run:

```
$ python -m pytest path/to/your/test.py
```

You can also run all unit tests under a specific directory:

```
$ python -m pytest tests/chainer_tests/<directory name>
```

Some tests require CUDA and cuDNN by default. In order to run unit tests that do not require CUDA and cuDNN, set an environment variable and filter using test marks as follows:

```
$ export CHAINER_TEST_GPU_LIMIT=0
$ python -m pytest path/to/your/test.py -m='not cudnn'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs to `CHAINER_TEST_GPU_LIMIT`. For example, if you only have a single GPU, launch `pytest` with the following command to skip multi-GPU tests:

```
$ export CHAINER_TEST_GPU_LIMIT=1
$ python -m pytest path/to/gpu/test.py
```

Some tests spend too much time. If you want to skip such tests, pass `-m='not slow'` option to the command:

```
$ python -m pytest path/to/your/test.py -m='not slow'
```

10.3.2 Test File and Directory Naming Conventions

Tests are found in the tests/chainer_tests directory. In order to enable the test runner to find test scripts correctly, we are using a special naming convention for the test subdirectories and the test scripts.

- The name of each subdirectory of `tests` must end with the `_tests` suffix.
- The name of each test script must start with the `test_` prefix.

When we write a test for a module, we use the appropriate path and file name for the test script whose correspondence to the tested module is clear. For example, if you want to write a test for a module `chainer.x.y.z`, the test script must be located at `tests/chainer_tests/x_tests/y_tests/test_z.py`.

10.3.3 How to Write Tests

There are many examples of unit tests under the tests directory, so reading some of them is a good and recommended way to learn how to write tests for Chainer. They use the `unittest` package of the standard library, while some tests are additionally using utilities from *`chainer.testing`*.

In addition to the *Coding Guidelines* mentioned above, the following rules apply to the test code:

- All test classes must inherit from `unittest.TestCase`.
- Use `unittest` features to write tests, except for the following cases:
  - Use `assert` statement instead of `self.assert*` methods (e.g., write `assert x == 1` instead of `self.assertEqual(x, 1)`).
  - Use `with pytest.raises(...):` instead of `with self.assertRaises(...):`.

Note: We are incrementally applying the above style. Some existing tests may be using the old style (`self.assertRaises`, etc.), but all newly written tests should follow the above style.

Even if your patch includes GPU-related code, your tests should not fail without GPU capability. Test functions that require CUDA must be tagged with the `chainer.testing.attr.gpu` decorator:

```
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.gpu
    def test_my_gpu_func(self):
        ...
```

The functions tagged with the gpu decorator are skipped if CHAINER_TEST_GPU_LIMIT=0 environment variable is set. We also have the chainer.testing.attr.cudnn decorator to let pytest know that the test depends on cuDNN. The test functions decorated with cudnn are skipped if -m='not cudnn' is given.

The test functions decorated with gpu must not depend on multiple GPUs. In order to write tests for multiple GPUs, use the chainer.testing.attr.multi_gpu() decorator instead:

```
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.multi_gpu(2)  # specify the number of required GPUs here
    def test_my_two_gpu_func(self):
        ...
```

If your test requires too much time, add the chainer.testing.attr.slow decorator. The test functions decorated with slow are skipped if -m='not slow' is given:

```
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
    ...

    @attr.slow
    def test_my_slow_func(self):
        ...
```

Note: If you want to specify more than two attributes, use and operator like -m='not cudnn and not slow'. See detail in the documentation of pytest.

10.4 Documentation

When adding a new feature to the framework, you should also document it in the reference so that other users can find it in the official documentation. For example, if you are adding a new function under chainer.functions, *Functions* should be updated.

The documentation source is stored under docs directory and written in reStructuredText format.

To build the documentation, you need to install Sphinx:

```
$ pip install sphinx sphinx_rtd_theme
```

Note: Docstrings (documentation comments in the source code) are collected from the installed Chainer module. If you have edited docstrings in checked-out source files and want to see those changes reflected in the generated html, Chainer must be installed in develop mode to see those changes reflected in the generated documentation. To do this use `pip install -e .` from the the top of the Chainer directory.

Then you can build the documentation in HTML format locally:

```
$ cd docs
$ make html
```

HTML files are generated under `build/html` directory. Open `index.html` with the browser and see if it is rendered as expected.

Note: If you are unsure about how to write the documentation or failed to build it locally, you can submit a PR without documentation. Reviewers will help you with it.

10.5 Other Forms of Contribution

There are several other ways in which you can contribute to Chainer without directly working with the code base. Following are such contributions.

- Sending a question/reply to StackOverflow (with `chainer` tag) or Chainer User Group
- Open-sourcing an external example
- Writing a post about Chainer

10.6 Development Cycle

This section explains the development process of Chainer.

10.6.1 Versioning

The versioning of Chainer follows PEP 440 and a part of Semantic versioning. The version number consists of three or four parts: `X.Y.Zw` where `X` denotes the **major version**, `Y` denotes the **minor version**, `Z` denotes the **revision number**, and the optional `w` denotes the pre-release suffix. While the major, minor, and revision numbers follow the rule of semantic versioning, the pre-release suffix follows PEP 440, the Python community standards.

Note that a major update basically does not contain compatibility-breaking changes from the last release candidate (RC). This is not a strict rule, though; if there is a critical bug in the API that need to be fixed for the major version, breaking changes may be introduced.

For more on backward compatibility, please refer to the *API Compatibility Policy*.

10.6.2 Release Cycle

Two tracks with different versions are developed in parallel. The first track is the **stable versions**, which is a series of minor (occasional revision) updates for the latest major version. The second track is the **development versions**, which is a series of pre-releases for the upcoming major version.

If `X.0.0` is the latest major version, followed by `Y.0.0` and `Z.0.0`, a typical release cycle timeline would be as follows.

Date	ver X	ver Y	ver Z
0 weeks	X.0.0rc1	–	–
4 weeks	X.0.0	Y.0.0a1	–
8 weeks	X.1.0*	Y.0.0b1	–
12 weeks	X.2.0*	Y.0.0rc1	–
16 weeks	–	Y.0.0	Z.0.0a1

(* These might be revision releases)

The dates shown in the left-most column are relative to the release of `X.0.0rc1`. In particular, each revision/minor release is made around four weeks after the previous one of the same major version, and the pre-release of the upcoming major version is made at the same time. Whether these releases are revision or minor is determined based on the contents of each update.

Note that there are only three stable releases for the versions `X.x.x` in the example table above. The number of stable releases may vary depending the development status of the following in this case `Y` and its number of required beta versions (a `b` followed by a number). During the parallel development of `Y.0.0` and `Z.0.0a1`, the version `Y` is treated as an **almost-stable version** and `Z` is treated as a development version.

If there is a critical bug found in `X.x.x` after stopping the development of version `X`, we may release a hot-fix for this version at any time.

A milestone for each upcoming release is published on GitHub. The GitHub milestones are used to group issues and PRs belonging to a release.

10.6.3 Git Branches

The `master` branch is used to develop pre-release versions. It means that **alpha, beta, and RC updates are developed at the `master` branch**. This branch contains the most up-to-date source tree that includes features newly added after the latest major version.

The stable version is developed on the `vN` branch where "N" reflects the version number (*versioned branch*). For example, v3.0.0, v3.1.0, and v3.2.0 are developed on the `v3` branch.

A PR from a contributor should in general be targeting the `master` branch. If the change can and should be applied to the stable version in addition, a member from the core team will make sure it is backported to be included in the next revision update.

If the change is only applicable to the stable version and not to the `master` branch, please send it to the versioned branch. We basically only accept changes to the latest versioned branch (where the stable version is developed) unless the fix is critical.

If you want to introduce a new feature in the `master` branch to the current stable version, please send a *backport PR* to the stable version (the latest `vN` branch). See the next section for details.

Note: a change that can be applied to both branches should be sent to the `master` branch.

10.6.4 Feature Backport Pull Requests

In general, new features in the development branch are not backported to the stable versions. If such backports can be motivated and are necessary however, they are welcomed. In such a case, a backport PR must be sent to the latest `vN` branch. **Note that we do not accept any feature backport PRs to older versions because we are not running quality assurance workflows (e.g. CI) for older versions which means that we cannot ensure that the PR is correctly ported.**

There are some rules on sending backport PRs.

- Prefix the PR title with **[backport]**.
- Include the original PR number in the PR description, e.g. "This is a backport of #XXXX".
- (Optional) Write in the PR description, the motivation behind the backport.

There is a backport tool maintained by the core team that automates the process of creating backport PRs conforming to the rules above.

Note: PRs that do not include any changes/additions to APIs (e.g. bug fixes, documentation improvements) are backported by the core team, but contributors are also welcome to do so for faster development.

CHAPTER

ELEVEN

TIPS AND FAQS

11.1 It takes too long time to compile a computational graph. Can I skip it?

Chainer does not compile computational graphs, so you cannot skip it, or, I mean, you have already skipped it :).

It seems you have actually seen on-the-fly compilations of CUDA kernels. CuPy compiles kernels on demand to make kernels optimized to the number of dimensions and element types of input arguments. Pre-compilation is not available, because we have to compile an exponential number of kernels to support all CuPy functionalities. This restriction is unavoidable because Python cannot call CUDA/C++ template functions in generic way. Note that every framework using CUDA require compilation at some point; the difference between other statically-compiled frameworks (such as cutorch) and Chainer is whether a kernel is compiled at installation or at the first use.

These compilations should run only at the first use of the kernels. The compiled binaries are cached to the `$(HOME)/.cupy/kernel_cache` directory by default. If you see that compilations run every time you run the same script, then the caching is failed. Please check that the directory is kept as is between multiple executions of the script. If your home directory is not suited to caching the kernels (e.g. in case that it uses NFS), change the kernel caching directory by setting the `CUPY_CACHE_DIR` environment variable to an appropriate path. See CuPy Overview for more details.

11.2 MNIST example does not converge in CPU mode on Mac OS X

Note: Mac OS X is not an officially supported OS.

Many users have reported that MNIST example does not work correctly when using vecLib as NumPy backend on Mac OS X. vecLib is the default BLAS library installed on Mac OS X.

We recommend using other BLAS libraries such as OpenBLAS.

To use an alternative BLAS library, it is necessary to reinstall NumPy. Here are instructions to install NumPy with OpenBLAS using Conda.

```
$ conda install -c conda-forge numpy
```

Otherwise, to install NumPy without Conda, you may need to install NumPy from source.

Use Homebrew to install OpenBLAS.

```
$ brew install openblas
```

Uninstall existing NumPy installation

```
$ pip uninstall numpy
```

You'll to create a file called *.numpy-site.cfg* in your home (~/) directory with the following:

```
[openblas]
libraries = openblas
library_dirs = /usr/local/opt/openblas/lib
include_dirs = /usr/local/opt/openblas/include
```

Install NumPy from the source code

```
pip install --no-binary :all: numpy
```

Confirm NumPy has been installed with OpenBLAS by running this command:

```
$ python -c "import numpy; print(numpy.show_config())"
```

You should see the following information:

```
blas_mkl_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
openblas_info:
  libraries = ['openblas', 'openblas']
  library_dirs = ['/usr/local/opt/openblas/lib']
  language = c
  define_macros = [('HAVE_CBLAS', None)]
  runtime_library_dirs = ['/usr/local/opt/openblas/lib']
 ...
```

Once this is done, you should be able to `import chainer` without OpenBLAS errors.

For details of this problem, see issue #704.

11.3 How do I fix InvalidType error?

Chainer raises an `InvalidType` exception when invalid inputs are given to *Functions*. If you got `InvalidType`, generally you need to check if `dtype` and/or `shape` of inputs are valid for the function.

Here are some examples of `InvalidType` errors:

```
import chainer.functions as F
import numpy as np

x = np.arange(10) - 5
F.relu(x)
```

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType:
Invalid operation is performed in: ReLU (Forward)

Expect: in_types[0].dtype.kind == f
Actual: i != f
```

In this case, kind of in_types[0] (which means the first input to the function, x) is expected to be f (floating-point), whereas the input was i (signed integer). You need to cast the input appropriately before passing to the function (e.g., x.astype(np.float32)).

```
import chainer.functions as F
import numpy as np

x = np.ones((4, 4))
y = np.ones((3, 3))
F.concat([x, y])
```

```
Traceback (most recent call last):
...
chainer.utils.type_check.InvalidType:
Invalid operation is performed in: Concat (Forward)

Expect: in_types[0].shape[0] == in_types[1].shape[0]
Actual: 4 != 3
```

In this case, the function expects that x.shape[0] is equal to y.shape[0], but actually it was 4 and 3, respectively.

See *Type Checks* for the detailed behavior of type checking system in Chainer.

11.4 How do I accelerate my model using Chainer Backend for Intel Architecture?

Follow these steps to utilize Chainer Backend for Intel Architecture in your model.

11.4.1 Install Chainer Backend for Intel Architecture

The following environments are recommended by Chainer Backend for Intel Architecture.

- Ubuntu 14.04 / 16.04 LTS (64-bit) and CentOS 7 (64-bit)
- Python 2.7.6+, 3.5.2+, and 3.6.0+

On recommended systems, you can install Chainer Backend for Intel Architecture wheel (binary distribution) by:

```
$ pip install 'ideep4py<2.1'
```

Note: ideep4py v1.0.x is incompatible with v2.0.x, and is not supported in Chainer v5.0 or later.

11.4.2 Enable Chainer Backend for Intel Architecture Configuration

Currently Chainer Backend for Intel Architecture is disabled by default because it is an experimental feature. You need to manually enable it by changing chainer.config.use_ideep configuration to 'auto'. See *Configuring Chainer* for details.

The easiest way to change the configuration is to set environment variable as follows:

```
export CHAINER_USE_IDEEP="auto"
```

You can also use *chainer.using_config()* to change the configuration.

```
x = np.ones((3, 3), dtype='f')
with chainer.using_config('use_ideep', 'auto'):
    y = chainer.functions.relu(x)
print(type(y.data))
```

```
<class 'ideep4py.mdarray'>
```

11.4.3 Convert Your Model to Chainer Backend for Intel Architecture

You need to call `model.to_intel64()` (in the same way you call `model.to_gpu()` to transfer your link to GPU) to convert the link to Chainer Backend for Intel Architecture.

11.4.4 Run Your Model

Now your model is accelerated by Chainer Backend for Intel Architecture!

Please note that not all functions and optimizers support Chainer Backend for Intel Architecture acceleration. Also note that Chainer Backend for Intel Architecture will not be used depending on the shape and data type of the input data.

11.5 My training process gets stuck when using MultiprocessIterator

When you are using OpenCV somewhere in your code and the *MultiprocessIterator* is used in the training code, the training loop may get stuck at some point. In such situation, there are several workarounds to prevent the process got stuck.

1. Set the environment variable as follows: `OMP_NUM_THREADS=1`
2. Add `cv2.setNumThreads(0)` right after `import cv2` in your training script.
3. Use *MultithreadIterator* instead of *MultiprocessIterator*.

This problem is originally reported here: A training loop got stuck in a certain condition with multi-processing updater and opencv for Chainer and the discussion on related problems is still going here: OpenCV + Python multiprocessing breaks on OSX.

CHAPTER TWELVE

PERFORMANCE BEST PRACTICES

This guide explains some tips and advice for maximizing the performance of Chainer.

12.1 Use the Latest Version

It is generally recommended that you use the latest version of Chainer and its dependent libraries (CUDA, cuDNN, iDeep, etc.). Some of the new features and performance optimizations introduced in newer versions of dependent libraries may not be available in older versions of Chainer. Also, Chainer itself is incrementally being improved to provide better performance.

If you are using Chainer v4 or later, you can check the version configuration by:

```
chainer.print_runtime_info()
```

```
Chainer: 4.0.0
NumPy: 1.14.3
CuPy:
  CuPy Version          : 4.0.0
  CUDA Root             : /usr/local/cuda
  CUDA Build Version    : 9000
  CUDA Driver Version   : 9000
  CUDA Runtime Version  : 9000
  cuDNN Build Version   : 7100
  cuDNN Version         : 7100
  NCCL Build Version    : 2102
```

Generally, the Chainer team is maintaining the API between minor updates (e.g., v4.0 to v4.1) so that users can upgrade Chainer without modifying their code (see *API Compatibility Policy* for our policy). As for major updates, please refer to the *Upgrade Guide* to understand what should be done for migration.

12.2 Enable Hardware Accelerations

12.2.1 Using GPU

In most cases, running on GPU will give you better performance than on CPU. When using GPU, also make sure to install cuDNN, which is a library to accelerate deep neural network computations.

Note: You don't have to manually install cuDNN if you are using CuPy wheels, which includes the latest version of cuDNN. Check the output of `chainer.print_runtime_info()`; if you see the cuDNN version number, it is installed properly and will be used by Chainer automatically.

Note: If you wish, you can manually disable use of cuDNN using `chainer.config.use_cudnn` configuration option. See *Configuring Chainer* for details.

12.2.2 Using CPU

If you are running Chainer on CPU, you can use iDeep to utilize vector instructions of CPU. See *Tips and FAQs* for steps to run your model with iDeep.

You can also improve performance by building NumPy linked to Intel MKL. See Numpy/Scipy with Intel® MKL and Intel® Compilers for the detailed instructions.

Note: If you installed numpy package using Anaconda, you may already have MKL-linked NumPy. Check the output of `numpy.show_config()` to see what linear algebra library is linked.

Note: Use of iDeep and MKL-linked NumPy are orthogonal. You can use both of them at once to maximize the performance.

12.3 Migrate Data Preprocessing Code from NumPy to CuPy

If you are preprocessing your dataset or running data augmentation using NumPy, you may be able to use CuPy as a substitution to improve performance.

Note: It is **not always** efficient to use CuPy instead of NumPy, especially when the computation is not very heavy, or it cannot be done in batch.

12.4 Avoid Data Transfer

If you are using GPU, be aware of data transfer between CPU and GPU. For example, `print`ing `chainer.Variable` on GPU (e.g., for debugging) will cause memory transfer from GPU to CPU, which will incur synchronization overhead.

You can use NVIDIA Visual Profiler to diagnose this kind of issue.

12.5 Optimize cuDNN Convolution

12.5.1 Workspace Size

Some convolution algorithms in cuDNN use additional GPU memory as a temporary buffer. This is called "workspace," and users can adjust the upper limit of its size. By increasing the limit of workspace size, cuDNN may be able to use better (i.e., memory consuming but faster) algorithm.

The default size (in bytes) is:

```
>>> chainer.backends.cuda.get_max_workspace_size()
8388608
```

and can be adjusted using *chainer.backends.cuda.set_max_workspace_size()*.

Maximum required workspace size may vary depending on various conditions such as GPU hardware and batch size of inputs.

12.5.2 Auto-Tuner

Some convolution algorithms in cuDNN support the auto-tuner feature that finds the fastest convolution algorithm for given inputs. You can turn on this feature by setting `autotune` configuration to `True`.

See *Configuring Chainer* for detailed descriptions.

Note: Auto-tuner tries to find the best algorithm for every first observation of the input shape combination. Therefore, the first batch will become slower when auto-tuner is enabled. The result of auto-tuner is cached on memory so that it can be reused for data with the same input shape combination. In other words, algorithm selected in the first batch will be reused for the second and later batches, as long as the input shape combination is the same.

If you set `autotune` configuration to `False`, the default convolution algorithm will always be selected, regardless of the previous auto-tuner results.

Note: Auto-tuner always use the maximum workspace size.

12.6 Fine-Tune Configuration

There are some Chainer configuration values that affect performance. Although the default values work well in most cases, you can adjust the following configurations for better performance.

- `enable_backprop`

  If you are running your model for inference (i.e., you don't have to use back propagation because you are not training the model), you can set this configuration to `False` to improve performance and reduce memory consumption.

- `type_check`

  By default, Chainer checks the integrity between input data and functions. This makes possible to display friendly message when, for example, data with invalid dtype or shape is given to a function. By setting this configuration to `False`, you can let Chainer skip such check to improve performance. It is recommended that you turn off the check only for well-tested code and input data.

See *Configuring Chainer* for detailed descriptions.

12.7 Load Datasets Concurrently

If loading process of your dataset is I/O-bound or CPU-bound, consider using *chainer.iterators.MultithreadIterator* or *chainer.iterators.MultiprocessIterator* to load dataset concurrently using multiple threads or processes, instead of `chainer.iterators.SerialIterator` which works in a single thread in a single process.

12.8 Use Multiple GPUs

You can utilize multiple GPUs to make the training process faster.

For data parallelism, you can use `chainer.training.updaters.ParallelUpdater` or `chainer.training.updaters.MultiprocessParallelUpdater` instead of `chainer.training.updaters.StandardUpdater`. For model parallelism, you need to manually transfer each `chainer.Link` in your model to each device.

See *Using GPU(s) in Chainer* for the working examples of each case.

12.9 Use Multiple Nodes

You can scale-out the training process of your Chainer model to multiple-node cluster by using *ChainerMN* module which enables distributed deep learning.

CHAPTER

THIRTEEN

UPGRADE GUIDE

This is a list of changes introduced in each release that users should be aware of when migrating from older versions. Most changes are carefully designed not to break existing code; however changes that may possibly break them are highlighted with a box.

13.1 Chainer v7

13.1.1 Dropping Support of Python 2.7

In Chainer v7, Python 2.7 is no longer supported as it reaches its end-of-life (EOL) in January 2020. Python 3.5.1 is the minimum Python version supported by Chainer v7. Please upgrade the Python version if you are using Python 2.7 to any later versions listed under *Installation*.

13.1.2 CuPy v7

Chainer v7 requires CuPy v7 if you need GPU support. Please see the Upgrade Guide for CuPy v7 for details.

13.2 Chainer v6

13.2.1 Dropping Support of Python 3.4

In Chainer v6, Python 3.4 is no longer supported as it reaches its end-of-life (EOL) in March 2019. Python 3.5.1 is the minimum Python 3 version supported by Chainer v6. Please upgrade the Python version if you are using Python 3.4 to any later versions listed under *Installation*.

13.2.2 CuPy Needs To Be Manually Updated

Prior to Chainer v6, CuPy is automatically updated to the appropriate version when updating Chainer (i.e., `pip install -U chainer` updates CuPy package). In Chainer v6, Chainer does not perform this automatic update. You need to manually update CuPy package when updating Chainer package.

This is because the automatic update made users difficult to switch between CuPy packages (e.g. `cupy-cuda90` and `cupy-cuda92` etc). See #5425 for details.

13.2.3 Deprecation Notice on Communicators and Old NCCL versions

Chainer v6 only supports NCCL 2.3 and newer versions. Old NCCL versions are to be deprecated and will be removed in future versions. As of old NCCL deprecation, several communicators built for them are to be deprecated as well:

- `hierarchical`
- `two_dimensional`
- `single_node`

They will be removed in future versions. Also, default communicator changed to *pure_nccl* from *hierarchical*.

13.2.4 CuPy v6

Chainer v6 requires CuPy v6 if you need GPU support. Please see the Upgrade Guide for CuPy v6 for details.

13.3 Chainer v5

13.3.1 ChainerMN Became Part of Chainer

ChainerMN, which enables multi-node distributed deep learning using Chainer, has been merged to Chainer v5.

Prior to Chainer v4, ChainerMN was provided as a separate `chainermn` package. In Chainer v5, ChainerMN now became a part of Chainer; ChainerMN will be installed just by installing `chainer` package. If you are using `chainermn` package, make sure to remove it by `pip uninstall chainermn` before upgrading to Chainer v5 or later.

For documentation of ChainerMN, see *Distributed Deep Learning with ChainerMN*.

13.3.2 Use `forward` Instead of `__call__` in Links

Prior to Chainer v5, `__call__` method is used to define the behavior of *`Link`*. In Chainer v5, `forward` method has been introduced, and is now recommended that you use it instead of `__call__`. The base class (*`Link`*) provides `__call__` method implementation that invokes `forward` method defined in the subclass; the only thing you need to do is to rename the method name (replace `def __call__(...)` with `def forward(...)`).

For backward compatibility, you can still use `__call__` to define your own link. However, new features introduced in Chainer v5 (e.g., *`LinkHook`*) may not be available for such links.

13.3.3 FunctionNode Classes are Hidden from `chainer.functions`

Prior to Chainer v5, *`FunctionNode`* classes (e.g., `chainer.functions.MaxPooling2D`) are exposed under *`chainer.functions`*. In Chainer v5, these classes are hidden from *`chainer.functions`*. Use the equivalent wrapper functions listed in *Functions* (e.g., *`chainer.functions.max_pooling_2d()`*) instead.

Some wrapper functions now provide options to access internal states to avoid directly using *`FunctionNode`* classes.

- *`chainer.functions.max_pooling_2d()`*: `return_indices`
- *`chainer.functions.max_pooling_nd()`*: `return_indices`
- *`chainer.functions.dropout()`*: `mask`, `return_mask`
- *`chainer.functions.gaussian()`*: `eps`, `return_eps`

For example, suppose your existing code needs to access `MaxPooling2D.indexes` to later perform upsampling:

```
p = F.MaxPooling2D(2, 2)
h = p.apply((x,))[0]
...
y = F.upsampling_2d(h, p.indexes, ksize=2)
```

The above code may raise this error in Chainer v5:

```
AttributeError: module 'chainer.functions' has no attribute 'MaxPooling2D'
```

You can rewrite the above code using `return_indices` option of *chainer.functions.max_pooling_2d()*:

```
h, indices = F.max_pooling_2d(x, 2, 2, return_indices=True)
...
y = F.upsampling_2d(h, indices, ksize=2)
```

13.3.4 Persistent Values are Copied in Link.copyparams

chainer.Link.copyparams() is a method to copy all parameters of the link to another link. This method can be used, for example, to copy parameters between two chains that partially share the same network structure to reuse pretrained weights.

Prior to Chainer v5, only parameters are copied between links. In Chainer v5, in addition to parameters, persistent values (see *Serializers – saving and loading* for details) are also copied between links. This is especially beneficial when copying parameters of *BatchNormalization*, as it uses persistent values to record running statistics.

You can skip copying persistent values by passing newly introduced `copy_persistent=False` option to *copyparams()* so that it behaves as in Chainer v4.

13.3.5 Updaters Automatically Call Optimizer.new_epoch

This change should affect only a minority of users (who call *new_epoch()* while using a trainer, or who implement their own *Updater* class).

Optimizers provide *new_epoch()* method, which can be used to change the behavior of optimizers depending on the current epoch number. Prior to Chainer v5, this method was expected to be called by users. In Chainer v5, updaters have been changed to call *new_epoch()* automatically. If you have been calling *new_epoch()* method manually while using a trainer (or an updater), you may need any of the following fixes:

- Pass `auto_new_epoch=False` to the constructor of the updater (e.g., *StandardUpdater*) to stop *new_epoch()* from being called automatically by the updater.
- Avoid calling *new_epoch()* method manually.

If you implement your own *Updater* class, you may need to update your code to automatically call *new_epoch()* (you can refer to the changes introduced in #4608 to understand how to fix your updater).

13.3.6 Extending the Backend Namespace

In addition to `chainer.backends`, we introduced `chainer.backend`. This subpackage contains utility functions that span several backends. For instance, it includes *chainer.backend.get_array_module()* which used to be defined in *chainer.backends.cuda.get_array_module()*. Both can be used but the latter will be deprecated.

13.3.7 `get_device_from_array` Returns Actual Device for Empty Arrays

Prior to Chainer v5, *chainer.backends.cuda.get_device_from_array()* returned `chainer.backends.cuda.DummyDeviceType` if the array is empty. In Chainer v5, it has been changed to return the actual `cupy.cuda.Device` object:

```
>>> x = cupy.array([])
>>> chainer.backends.cuda.get_device_from_array(x)
<CUDA Device 0>
```

13.3.8 Update of Docker Images

Chainer official Docker images (see *Installation* for details) are now updated to use CUDA 9.2 and cuDNN 7.

To use these images, you may need to upgrade the NVIDIA driver on your host. See Requirements of nvidia-docker for details.

13.3.9 CuPy v5

Chainer v5 requires CuPy v5 if you need GPU support. Please see the Upgrade Guide for CuPy v5 for details.

13.4 Chainer v4

13.4.1 Introduction of Backend Namespace

We introduced `chainer.backends` subpackage for future support of various backend libraries other than NumPy and CuPy. By this change, `chainer.cuda` module is now moved to `chainer.backends.cuda`.

This does not break the existing code; you can safely continue to use `chainer.cuda` (e.g., `from chainer import cuda`) but it is now encouraged to use `from chainer.backends import cuda` instead.

13.4.2 Namespace Changes for Updaters

`chainer.training.StandardUpdater` and `chainer.training.ParallelUpdater` are now moved to *chainer.training.updaters.StandardUpdater* and *chainer.training.updaters.ParallelUpdater* respectively, to align with the namespace convention of other subpackages. See the discussion in #2982 for more details.

This change does not break the existing code; you can safely continue to use updater classes directly under `chainer.training` but it is now encouraged to use `chainer.training.updaters` instead.

13.4.3 Namespace Changes for Optimizer Hooks

Optimizer hook functions are moved from `chainer.optimizer.*` to `chainer.optimizer_hooks.*`. For example, `chainer.optimizer.WeightDecay` is now located *chainer.optimizer_hooks.WeightDecay*.

If the existing code is using hooks directly under `chainer.optimizer`, `DeprecationWarning` will be shown. You are now encouraged to use `chainer.optimizer_hooks` instead.

13.4.4 Prohibition of Mixed Use of Arrays on Different Devices in Function Arguments

Argument validation of functions is now strictened to check device consistency of argument variables to provide better error messages to users. Suppose the following code:

```
v1 = chainer.Variable(np.arange(10, dtype=np.float32))      # CPU
v2 = chainer.Variable(cupy.arange(10, dtype=cupy.float32))  # GPU

# The line below raises an exception, because arguments are on different device.
F.maximum(v1, v2)
```

Prior to v4, the above code raises an exception like `ValueError: object __array__ method not producing an array`, which was difficult to understand. In v4, the error message would become `TypeError: incompatible array types are mixed in the forward input (Maximum)`. This kind of error usually occurs by mistake (for example, not performing `to_gpu` for some variables).

> **Attention:** As the argument validation is strictened, call of functions intentionally mixing NumPy/CuPy arrays in arguments will not work in Chainer v4. Please transfer all arrays to the same device before calling functions.

13.4.5 References to Function Nodes Not Retained in TimerHook and CupyMemoryProfilerHook

To reduce memory consumption, references to the function nodes will no longer be retained in the *chainer.function_hooks.CupyMemoryProfileHook* and *chainer.function_hooks.TimerHook*. See the discussion in #4300 for more details.

> **Attention:** The existing code using function nodes retained in `call_history` attribute of these hooks will not work. The first element of `call_history` became the name of the function, instead of the function node instance itself. You can define your own function hook if you need to access the function node instances.

13.4.6 Update of Docker Images

Chainer official Docker images (see *Installation* for details) are now updated to use CUDA 8.0 and cuDNN 6.0. This change was introduced because CUDA 7.5 does not support NVIDIA Pascal GPUs.

To use these images, you may need to upgrade the NVIDIA driver on your host. See Requirements of nvidia-docker for details.

13.4.7 CuPy v4

Chainer v4 requires CuPy v4 if you need GPU support. Please see the Upgrade Guide for CuPy v4 for details.

13.5 Chainer v3

13.5.1 Introduction of New-style Functions

This release introduces new-style functions (classes inheriting from `FunctionNode`) that support double backward (gradient of gradient). See the Release Note for v3.0.0 for the usage of this feature.

Many of *Functions* are already migrated to new-style, although some of functions are still old-style (classes inheriting from `Function`). We are going to migrate more old-style functions to new-style in upcoming minor releases.

This does not break the existing code. Old-style functions (classes inheriting from `Function`) are still supported in v3 and future versions of Chainer.

If you are going to write new functions, it is encouraged to use `FunctionNode` to support double backward.

> **Attention:** Users relying on undocumented function APIs (directly instantiating old-style classes) may experience an error like `TypeError: 'SomeFunction' object is not callable` after upgrading to v3. Please use the function APIs documented in *Functions*.

13.5.2 Changed Behavior of matmul Function

The behavior of `chainer.functions.matmul()` has been changed to behave like the corresponding NumPy function (`numpy.matmul()`). See the discussion in #2426 for more details.

> **Attention:** The existing code using `chainer.functions.matmul()` may require modification to work with Chainer v3.

Also note that `chainer.functions.batch_matmul()` is now deprecated by this change. You can rewrite it using `chainer.functions.matmul()`.

13.5.3 Removed use_cudnn Argument in spatial_transformer_grid and spatial_transformer_sampler Functions

`use_cudnn` argument has been removed from `chainer.functions.spatial_transformer_grid()` and `chainer.functions.spatial_transformer_sampler()`. See the discussion in #2955 for more details.

> **Attention:** The existing code using `use_cudnn` argument of `chainer.functions.spatial_transformer_grid()` and `chainer.functions.spatial_transformer_sampler()` require modification to work with Chainer v3. Please use the configuration context (e.g., `with chainer.using_config('use_cudnn', 'auto'):`) to enable or disable use of cuDNN. See *Configuring Chainer* for details.

13.5.4 CuPy v2

Chainer v3 requires CuPy v2 if you need GPU support. Please see the Upgrade Guide for CuPy v2 for details.

13.6 Chainer v2

See *Upgrade Guide from v1 to v2* for the changes introduced in Chainer v2.

13.6.1 Upgrade Guide from v1 to v2

This documentation provides detailed information of differences between Chainer v1 and v2. You will know by reading it which part of your code is required (or recommended) to be fixed when you upgrade Chainer from v1 to v2.

CuPy

CuPy has been separated from Chainer into a separate package

CuPy, which was originally a part of Chainer, has been separated into a different Python package since Chainer v2. It changes the way to set up Chainer with CUDA support. In particular, you have to separately install `cupy` package to enable CUDA support. See *Installation* for the recommended installation steps.

Fortunately, there is no need of updating your source code to catch up with this change.

Global configurations

Training mode is configured by a thread-local flag

In Chainer v2, the concept of *training mode* is added. It is represented by a thread-local flag `chainer.config.train`, which is a part of *the unified configuration*. When `chainer.config.train` is `True`, functions of Chainer run in the training mode, and otherwise they run in the test mode. For example, `BatchNormalization` and `dropout()` behave differently in each mode.

In Chainer v1, such a behavior was configured by the `train` or `test` argument of each function. **This train/test argument has been removed in Chainer v2.** If your code is using the `train` or `test` argument, you have to update it. In most cases, what you have to do is just removing the `train` / `test` argument from any function calls.

Example

Consider the following model definition and the code to call it in test mode written for Chainer v1.

```
# Chainer v1
import chainer.functions as F

```

(continues on next page)

(continued from previous page)

```
class MyModel(chainer.Link):
    ...

    def __call__(self, x, train=True):
        return f(F.dropout(x, train=train))

m = MyModel(...)
y = m(x, train=False)
```

In Chainer v2, it should be updated into the following code:

```
# Chainer v2
import chainer.functions as F

class MyModel(chainer.Link):
    ...

    def __call__(self, x):
        return f(F.dropout(x))

m = MyModel(...)
with chainer.using_config('train', False):
    y = m(x)
```

Configurations are added and replace some of existing global flags

There are many global settings moved to *the unified configuration* other than the training mode. Following is the complete list of the configuration entries that have corresponding features in Chainer v1.

chainer.config.cudnn_deterministic It is corresponding to the `deterministic` argument of some convolution functions in Chainer v1. **This argument has been removed since Chainer v2.** If you are using this argument, you have to use the `chainer.config.cudnn_deterministic` flag to change the behavior of the convolution functions.

chainer.config.debug It is corresponding to the debug mode in Chainer v1, which was configured by `set_debug()` and extracted by `is_debug()`. These functions are also available in Chainer v2, so you basically do not need to update the code related to the debug mode.

chainer.config.enable_backprop It is corresponding to the *backprop mode* in Chainer v1. The functions `no_backprop_mode()` and `force_backprop_mode()` are still available in Chainer v2, which automatically turns on/off the `enable_backprop` flag. One important difference from Chainer v1 is that **the** `volatile` **flag is removed from** `Variable`. Therefore, there are more situations that you need to modify the `enable_backprop` flag.

chainer.config.keep_graph_on_report This flag configures whether or not to keep the computational graph alive for a reported variable. In Chainer v2, when a `Variable` object is reported by `report()`, a copy of the variable isolated from the computational graph is created and stored by default. Setting `True` to this flag, you can change this behavior and then the original `Variable` object is stored as is. See *When a variable is reported, the variable is copied with the graph purged* for the details.

chainer.config.train It is corresponding to the `train` or `test` argument of some functions in Chainer v1. **This argument has been removed since Chainer v2.** If you are using this argument, you have to use the `chainer.config.train` flag instead. See *Training mode is configured by a thread-local flag* for more details.

chainer.config.type_check It is corresponding to the `Function.type_check_enable` flag. If your code touches this flag, **you have to use** `chainer.config.type_check` **instead.** Note that the environment variable `CHAINER_TYPE_CHECK` is still available in Chainer v2, so if you are only using the environment variable, there is no need of updating your code.

chainer.config.use_cudnn It is corresponding to the `use_cudnn` argument of many functions that have cuDNN implementations. **This argument has been removed since Chainer v2.** If you are using this argument, you have to use the `chainer.config.use_cudnn` flag instead. *Note that this flag is ternary, not binary.* See *Configuring Chainer* for more details.

These configurations can be modified in two ways.

- Simply substituting a new value to an entry, like `chainer.config.train = False`.
- Using the `chainer.using_config` context manager. It can be used with the `with` statement of Python as follows:

  ```
  with chainer.using_config('train', False):
      do something  # this code runs with chainer.config.train == False
  ```

  It recovers the original configuration after quitting the `with` block.

The `chainer.config` manages *the thread-local configuration*. You can also set the global configuration by modifying `chainer.global_config`. Note that the global configuration is used only if the entry of the thread-local configuration is not explicitly set up.

Variable

Volatile flag is removed

The `Variable.volatile` flag has been removed since Chainer v2.

Instead, the configuration `chainer.config.enable_backprop` can be used to enable/disable the automatic differentiation feature. If it is `True`, Chainer always creates a computational graph on the forward propagation, which corresponds to passing non-volatile variables in Chainer v1. Otherwise, Chainer does not create a graph, which corresponds to passing volatile variables in Chainer v1. The biggest difference is that `enable_backprop` is a thread-local flag, whereas `volatile` was a flag local to each *`Variable`* object. Note that `enable_backprop` flag has already existed in Chainer v1, which took effect only if all the inputs to the function have `volatile == 'auto'`.

The `chainer.config.enable_backprop` flag can be modified directly or by using *`using_config()`*. See *Configuring Chainer* for details. There is also a convenience function, *`no_backprop_mode()`*, to turn off the flag.

If you are using the `Variable.volatile` flag, you have to stop setting this flag (it will not take effect), and set the `enable_backprop` flag instead.

Example

Let `model` be your model, and consider the following code that calls it in volatile mode.

```
# Chainer v1
x_data = ...   # ndarray
x = chainer.Variable(x_data, volatile=True)
y = model(x)
```

In Chainer v2, it should be updated as follows.

```
# Chainer v2
x_data = ...    # ndarray
x = chainer.Variable(x_data)
with chainer.no_backprop_mode():
    y = model(x)
```

Variable is not a part of a computational graph anymore

The *Variable* class has been separated into two distinct classes, the *Variable* class and the `VariableNode` class, since Chainer v2. Every *Variable* object owns its own `VariableNode` object. A computational graph consists of *Function* objects and `VariableNode` objects. When one applies a *Function* to a *Variable*, the `VariableNode` object of the variable is extracted and set to one of the inputs of the function.

Note that the underlying data array of the variable is still held by the *Variable* object. It allows each *Function* implementation to release unneeded arrays from the computational graph, resulting in greatly reduced memory consumption.

This change does not affect most users' code. If you are directly traversing the computational graph by yourself or modifying the graph ad-hoc, you may have to update your code. In most cases, it is enough to just change *Variable* into `VariableNode` in the code traversing the computational graph.

Parameter has to be an instance of Parameter class

Chainer v2 has a subclass of *Variable* called *Parameter*. This class has an interface convenient on setting up a parameter variable registered to *Link*.

You basically do not need to update your code because *Link.add_param()* creates a *Parameter* object in Chainer v2. There is a new recommended way of registering parameters to a link in Chainer v2, though. *See here* for the recommended way of parameter registration.

Small changes to Variable

There are some changes on the interface and specification of methods.

- `len(variable)` returns the length of the first axis of the underlying array in Chainer v2. This is equivalent to `len(variable.data)`. It is different from the behavior of Chainer v1, in which `len` returned the total number of elements in the underlying array.
- `repr(variable)` returns a NumPy-like text representation of the underlying array in Chainer v2. In Chainer v1, it just returns a string that shows the name of the variable.

Function

The force_tuple option of split_axis is True by default

In Chainer v2, the `force_tuple` argument of *functions.split_axis()* is set to `True` by default. Therefore, it always returns a tuple regardless of the number of sections made after the split. It was `False` by default in Chainer v1.

Type check APIs are updated to enable lazy building of the error messages

In Chainer v2, the type check APIs are updated so that the overhead of checking types is greatly reduced. In order to achieve the overhead reduction, some APIs are changed.

If you have custom Function implementations that do type checking, you have to update your code. The following list shows which part has to be updated.

- Use `utils.type_check.eval()` instead of `Expr.eval`.
- Use `utils.type_check.make_variable()` to create a `utils.type_check.Variable` object instead of directly constructing it by yourself.
- Stop using `.name` attribute of any expression.

Background of this change: In Chainer v1, the type checking APIs build an abstract syntax tree (AST) based on each expression that tests some condition. The AST is used to emit a kind error message. However, building an AST requires constructions of many Python objects, which adds large Python overheads. In Chainer v2, the `Function.type_check_forward()` method is called once or twice. At the first call, the type checking APIs run in *light-weight mode*, where it does not build an AST and just checks the condition. The second call is made only if there is a test that fails, where it builds an AST. This change makes the ordinary path of running the type checking much faster, while keeping the kind error messages.

Methods to release unneeded arrays are added

As is written above, Chainer v2 introduced a new mechanism to reduce the memory consumption of each `Function` implementation. In many cases, a `Function` implementation does not need some input arrays in its backward computation. A new method called `Function.retain_inputs()` can be used to specify which input arrays are actually needed. This method must not be called from the outside of `Function.forward()`.

Example

For example, consider the following simple addition function.

```
class AddFunction(chainer.Function):
    def forward(self, inputs):
        return inputs[0] + inputs[1],

    def backward(self, inputs, grad_outputs):
        return grad_outputs[0], grad_outputs[0]
```

It can be seen that the backward computation of this function does not use any of the inputs. Then, specifying an empty tuple of indexes to `retain_inputs()` will reduce the memory overhead.

```
class AddFunction(chainer.Function):
    def forward(self, inputs):
        self.retain_inputs(())  # does not retain both inputs
        return inputs[0] + inputs[1],

    def backward(self, inputs, grad_outputs):
        return grad_outputs[0], grad_outputs[0]
```

In some cases, the function can (or have to) use the output arrays instead of the inputs in its backward computation. In Chainer v1, we have written code that store the output arrays to attributes of the `Function` object and reuse them in the `backward()` method. In Chainer v2, it is recommended that you use `Function.retain_outputs()`

to declare which outputs are required in the backward computation. The retained output arrays can be accessed via *Function.output_data*.

Note: The existing *Function* implementations that store the output arrays to its attributes will run correctly in Chainer v2. There is no any memory overhead right now. It is recommended that you use *retain_outputs()*, though, so that we can incorporate more memory optimization in the future.

Example

For example, consider the following simple implementation of the tanh function.

```
class TanhFunction(chainer.Function):
    def forward(self, inputs):
        xp = chainer.cuda.get_array_module(inputs[0])
        self.y = xp.tanh(inputs[0])
        return self.y,

    def backward(self, inputs, grad_outputs):
        one = self.y.dtype.type(1)  # avoid type promotion
        return grad_outputs[0] * (one - self.y * self.y),
```

We can use *retain_outputs()* instead of preserving the output array by ourselves as follows.

```
class TanhFunction(chainer.Function):
    def forward(self, inputs):
        self.retain_outputs((0,))
        xp = chainer.cuda.get_array_module(inputs[0])
        return xp.tanh(inputs[0]),

    def backward(self, inputs, grad_outputs):
        y = self.output_data[0]
        one = y.dtype.type(1)  # avoid type promotion
        return grad_outputs[0] * (one - y * y)
```

Link/Chain/ChainList

wscale option is removed from links

The `wscale` option has been removed from links since Chainer v2. **If you are using wscale option, you have to update your code.** The recommended way is to explicitly set the initializer.

Example

Consider the case of adding a *Linear* link with the weight initialized by 0.5x of the default initialization.

```
# Chainer v1
linear = chainer.links.Linear(10, 5, wscale=0.5)
```

Note that the default initializer of the weight matrix of *Linear* is a normal distribution of the standard deviation $1/\sqrt{fanin}$. Therefore, it can be fixed as follows.

```
# Chainer v2
linear = chainer.links.Linear(10, 5, initialW=chainer.initializers.Normal(0.5 / math.
↪sqrt(10)))
```

Or, by using the fact that `initializers.HeNormal` provides the initialization with a normal distribution of the standard deviation $scale * \sqrt{2/fanin}$, the following code is also equivalent to the original.

```
# Chainer v2, using HeNormal
linear = chainer.links.Linear(10, 5, initialW=chainer.initializers.HeNormal(0.5 /␣
↪math.sqrt(2))
```

bias option is removed from links

In Chainer v2, the `bias` option is removed from the following links: `Linear`, `Convolution2D`, `Deconvolution2D`, and `DilatedConvolution2D`. The effect of this argument was duplicated with the `initial_bias` option. Use `initial_bias` instead.

The bias vector is enabled by default in N-dimensional convolution links

In Chainer v2, the bias parameter is enabled by default in `ConvolutionND` and `DeconvolutionND`. It was unintentionally disabled by default in Chainer v1.

If you are using ConvolutionND or DeconvolutionND without specifying the `initial_bias` **argument, you have to fix your code.** If you want to keep the old behavior (i.e., no bias vector is created by the link), pass `nobias=True` to the link at the construction. Otherwise it will automatically create a bias vector.

init_weight function is removed

The `chainer.initializers.init_weight` function that was used on weight initialization has been removed since Chainer v2.

You have to update your code if you are using `init_weight`. In most cases, the update is simple: pass an initializer to `Parameter`.

Example

Consider the following code that initializes a weight matrix randomly and a bias vector by zero.

```
# Chainer v1
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__(
            W=(10, 5),
            b=(5,),
        )
        chainer.initializers.init_weight(self.W, chainer.initializers.Normal(0.05))
        self.b.data.fill(0)
    ...
```

This code should be fixed as follows (see the next topic for the use of `Parameter`).

```
# Chainer v2
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__()
        self.W = chainer.Parameter(chainer.initializers.Normal(0.05), (10, 5))
        self.b = chainer.Parameter(0, (5,))
    ...
```

The order of arguments of GRU is changed

In Chainer v2, the first two arguments of *GRU* is the input size and the output size. It was reversed in Chainer v1, causing an inconsistent interface compared to other links including *LSTM*. **If you are using** *GRU***, you have to update your code.** The update is done by simply flipping the first two arguments.

Example

Consider the following code that creates a *GRU* link.

```
# Chainer v1
gru = chainer.links.GRU(20, 10)
```

It should be fixed into the following code.

```
# Chainer v2
gru = chainer.links.GRU(10, 20)
```

Note that if you were omitting the output size, the code works as is because *GRU* supports *the omitted input size*.

```
# Chainer v1/v2
gru = chainer.links.GRU(20)
```

The default value of the forget bias for LSTM and StatelessLSTM is changed to 1

In Chainer v2, the default forget bias value of *LSTM* and *StatelessLSTM* links is changed to 1. This change is based on the paper reporting that using a large forget bias improves the training performance. The new behavior is also consistent with the implementation of BasicLSTMCell in TensorFlow.

It will improve the most use cases of LSTMs, although this change would break the reproducibility of the existing experiments. **If you want to keep the same initialization procedure, you have to update your code.** The change is simple: pass `forget_bias_init=0` to *LSTM* and *StatelessLSTM*.

The interfaces of GRU and LSTM are aligned

In Chainer v1, *GRU* was *stateless*, as opposed to the current implementation. To align with the naming convention of LSTM links, we have changed the naming convention from Chainer v2 so that the shorthand name points the stateful links. **If you are using** `StatelessGRU` for stateless version, whose implementation is identical to `chainer.linksGRU` in v1.

Aliases of links in chainer.functions are removed

For the compatibility reason, there were some links that have aliases in the `chainer.functions` module. These aliases are removed in Chainer v2. Use `chainer.links` instead.

Parameter link is removed

The `chainer.links.Parameter` link is removed in Chainer v2. This link existed in Chainer v1 only for the backward compatibility. Use `chainer.Parameter` instead (for the new `Parameter` class, see *Parameter has to be an instance of Parameter class*).

New-style parameter registration APIs are added to Link

In Chainer v2, `Link.init_scope()` method returns a context manager that automatically registers a `Parameter` object to the link at setting it to an attribute. If you are using IDE like PyCharm, it is recommended that you use this new-style parameter registration so that IDEs can easily detect the existence of the parameter as an attribute. It is also a good practice to use the new-style API even if you are not using IDEs, if you are planning to make the code public.

Note: The existing code that uses the conventional way of registering parameters are still valid.

Example

For example, the following link initialization code

```
# Chainer v1
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__(
            W=(10, 5),
            b=(5,),
        )
        chainer.initializers.Normal(0.05)(self.W.data)
        self.b.data.fill(0)
    ...
```

is recommended to be updated as follows.

```
# Chainer v2
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(chainer.initializers.Normal(0.05), (10, 5))
            self.b = chainer.Parameter(0, (5,))  # initialize by zero
    ...
```

Note: To keep a `Parameter` object as an attribute without registration, you can set the attribute without using the `with self.init_scope():` block.

New-style child link registration APIs are added to Chain

Like *Parameter*, a *Link* object is also automatically registered to a *Chain* object by substitution to an attribute within a *init_scope()* scope. If you are using IDE like PyCharm, it is recommended that you use the new-style child link registration so that IDEs can easily detect the existence of the child link as an attribute. It is also a good practice to use the new-style API even if you are not using IDEs, if you are planning to make the code public.

Note: The existing code that uses the conventional way of registering child links are still valid.

Example

For example, the following chain initialization code

```
# Chainer v1
class MyMLP(chainer.Chain):
    def __init__(self):
        super(MyMLP, self).__init__(
            layer1=L.Linear(None, 20),
            layer2=L.Linear(None, 30),
        )
    ...
```

is recommended to be updated as follows.

```
# Chainer v2
class MyMLP(chainer.Chain):
    def __init__(self):
        super(MyMLP, self).__init__()
        with self.init_scope():
            self.layer1 = L.Linear(20)
            self.layer2 = L.Linear(30)
```

Note that this example also demonstrates the use of new APIs with *the omitted input size*, explained below.

Note: To keep a *Link* object as an attribute without registration, you can set the attribute without using the `with self.init_scope():` block.

The input-size placeholder of links are made optional

In Chainer v2, the input size of many links, including *Linear* and *Convolution2D*, is made optional. In Chainer v1, we had to use `None` as the placeholder to specify that the input size should be determined at the first iteration. The placeholder can also be used in Chainer v2, although it is easier to just omit the input size.

See the previous item for the example of omitting the input size of *Linear*. The following links currently support the omitted input size.

- *Convolution2D*
- *Deconvolution2D*
- *DilatedConvolution2D*
- *Linear*

- *LSTM*
- *MLPConvolution2D*
- *StatelessLSTM*

Optimizer

Deprecated methods of Optimizer are removed

The following methods are removed from *Optimizer*. These methods have been already deprecated in the past versions. **If you are using these methods, you have to update your code.**

- `zero_grads`: use *Link.zerograds()* instead.
- `compute_grads_norm`: you can compute the gradient norm by iterating the list of parameters by *Link.params()*.
- `clip_grads`: use `GradientClipping` instead.
- `weight_decay`: use `WeightDecay` instead.
- `accumulate_grads`: use *Link.addgrads()* instead.

GradientMethod uses Link.cleargrads instead of Link.zerograds by default

In Chainer v2, *GradientMethod* clears the gradient before running backprop by *Link.cleargrads()*. It means that the gradient of each parameter is initialized by `None` instead of a zero array. Note that all the optimizer implementations provided by Chainer are subclasses of *GradientMethod*, and therefore this change affects all of them.

In most cases, you do not need to update your code. If your code relies on the zeroing initialization, you have to fix your code to explicitly initialize the gradient by zero, or to pass `False` to *GradientMethod.use_cleargrads()*.

GradientMethod is redesigned to allow parameter-specific update rules

In Chainer v2, the new class *UpdateRule* is used to define an update rule specific to each *Parameter* object. The *UpdateRule* is set to each *Parameter* object, and is used at each update step. This object implements an *update formula* using the data and gradient arrays.

Each *UpdateRule* object has `enabled` flag, which configures if the update rule should be applied to that parameter on update. By setting the flag to `False`, you can *freeze* the parameter. There is also a convenient method *Link.enable_update()* and *Link.disable_update()*, which configure the flag of each parameter under the link hierarchy. In other frameworks, a similar feature is called *layer freezing*. In Chainer v2, this is officially supported by these methods.

Each *UpdateRule* object can also hold its own hook functions similar to *Optimizer*. The built-in hook functions except for `GradientClipping` can also be used as a hook function of *UpdateRule*.

In most cases, you do not have to update your code because each optimizer automatically sets up an appropriate `UpdaterRule` object to each parameter.

If you are using a custom gradient-based optimizer implementation, you need to update the implementation. The following list shows what you have to do.

- Write a subclass of *UpdateRule* that implements the update rule.

- Rewrite your *GradientMethod* implementation. The new implementation only has to set up the update rule for each parameter in the target link.

You can see live examples in the optimizer implementations provided by Chainer.

Serializer

None is serializable

In Chainer v2, all serializers start supporting `None` value to be serialized and deserialized. Users' code can rely on this feature, i.e., it can serialize and deserialize `None` value with any given serializer. This change only affects your code if it provides its own serializer implementations.

Trainer and Extension

Updater and Evaluator pass raw data arrays to the loss function

In Chainer v2, *Updater* and *Evaluator* pass raw data arrays to the loss function without wrapping them with *Variable*. **You might need to update your code so that the loss function (in most cases, the model's `__call__`) accepts raw arrays.**

Note that raw arrays can be directly passed to any *Function*; they are automatically wrapped by *Variable*. For example, if the input is directly passed to a *Function* object (or any function under *chainer.functions*), you do not need to update the code.

Example

Consider the following code that obtains the shape of the input via *Variable.data*.

```
# Chainer v1
class MyLink(chainer.Link):
    def __call__(self, x):
        shape = x.data.shape  # valid if x is Variable, invalid if x is ndarray
        ...
```

It should be updated so that the link also accepts a raw array as the input. In this case, we have *Variable.shape* which is equivalent to `data.shape`, so you can simply write as follows.

```
# Chainer v2
class MyLink(chainer.Link):
    def __call__(self, x):
        shape = x.shape  # valid regardless of x being Variable or ndarray
        ...
```

---

trigger option is removed from snapshot and snapshot_object

In Chainer v2, the `trigger` option is removed from the *snapshot()* and `snapshot_object()` extensions. The effect of the option was duplicated with the `trigger` option of *Trainer.extend*. **If you are passing the** `trigger` **argument to these extensions, you have to update your code.** The update can be done by passing the value to the corresponding *Trainer.extend*.

Example

Assume that `trainer` is an instance of *Trainer*, and consider that you were adding a *snapshot()* extension as follows.

```
# Chainer v1
trainer.extend(chainer.training.extensions.snapshot(trigger=(1000, 'iteration')))
```

It should be updated as follows (note that this code also works with Chainer v1).

```
# Chainer v1/v2
trainer.extend(chainer.training.extensions.snapshot(), trigger=(1000, 'iteration'))
```

Extension.invoke_before_training is removed

In Chainer v2, The attribute `invoke_before_training` of *Extension* is removed. Instead, the *Extension.initialize* method is added. This method is called by *Trainer.run* before entering the training loop.

In Chainer v1, the extension is just called before entering the training loop when `invoke_before_training` is `True`. **If you have a custom extension that has** `invoke_before_training=True` **, you have to update the code.** What you have to do is to remove the `invoke_before_training` flag and override *initialize()* method. If you are using the *make_extension()* decorator, you can set the `initialize` function by passing the `initializer` argument to *make_extension()*.

The dump_graph extension dumps the valid graph only at its first invocation

In Chainer v2, the `dump_graph()` extension dumps the valid computational graph only at its first invocation. **If you want to dump the graph more than once, you have to fix the code.** The easiest fix is setting the `chainer.config.keep_graph_on_report` flag to `True`. *Note that this fix will cancel the improvement on the memory consumption made in Chainer v2.* More memory-efficient fix is to dump the graph without using an extension, e.g. by customizing the loss function or the updater.

Here is the background of this change. In Chainer v2, *the Reporter copies reported variables with purging the computational graph by default.* On the other hand, the `dump_graph()` extension requires the computational graph reachable from the reported variable. In order to make the graph available, the `dump_graph()` extension turns on the `chainer.config.keep_graph_on_report` flag at its initializer (i.e., it turns on the graph before entering the training loop). Since we also wanted to achieve the memory efficiency, the `dump_graph()` extension **turns off the flag after dumping the graph at its first invocation** (strictly speaking, it recovers the original value). As a result, the computational graph is not available from the second invocation.

Since the `dump_graph()` recovers the original flag value at its invocation, you can keep the graph dumped more than once by changing the original flag value.

Reporter

When a variable is reported, the variable is copied with the graph purged

In Chainer v2, when a *Variable* object is reported using *report()* function (or directly using *Reporter*), a copy of the variable is made without preserving the computational graph. **If your code depends on the reachability of the computational graph from the reported variable, you have to update your code.** The easiest way to update your code is setting `chainer.config.keep_graph_on_report` to `True`, then Chainer will keep the computational graph reachable from the reported variable.

The possible examples that are affected by this change are as follows (not exhaustive).

- A custom extension that runs backprop from a reported variable. It is definitely an example of assuming the reachability of the computational graph from the reported variable.
- An extension that visualizes the computational graph from a reported variable. If you are writing such an extension by yourself, you have to turn on the `keep_graph_on_report` flag. The `dump_graph()` extension is another example, for which see *the above item* for the details.

This change is made for the memory performance reason; with this change, the memory used by the computational graph for training is immediately released before invoking extensions. Therefore, *changing the behavior by overwriting* `chainer.config.keep_graph_on_report` *may increase the memory consumption.* It may cause an out-of-memory error if the computational graph of the loss function consumes almost all the memory available in your environment and there is an extension that uses a certain amount of memory (e.g. `Evaluator`).

Other utilities

Some obsolete classes and functions are removed

The following classes and functions are removed in Chainer v2.

- `chainer.Flag`
- `chainer.FunctionSet` (Use `Chain` or `ChainList` instead)
- `chainer.cuda.init` (It did nothing except for calling `check_cuda_available()`)
- `chainer.cuda.empty` (Use `cupy.empty()`)
- `chainer.cuda.empty_like` (Use `cupy.empty_like()`)
- `chainer.cuda.full` (Use `cupy.full()`)
- `chainer.cuda.full_like` (Use `cupy.full_like()`)
- `chainer.cuda.ones` (Use `cupy.ones()`)
- `chainer.cuda.ones_like` (Use `cupy.ones_like()`)
- `chainer.cuda.zeros` (Use `cupy.zeros()`)
- `chainer.cuda.zeros_like` (Use `cupy.zeros_like()`)

CHAPTER

FOURTEEN

LICENSE

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[LeCun98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324, 1998.

[Simonyan14] Simonyan, K. and Zisserman, A., Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1409.1556, 2014.

[He16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, 2016.

[Graves2006] Alex Graves, Santiago Fernandez, Faustino Gomez, Jurgen Schmidhuber, Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks

[Graves2012] Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z