Chainer

Release 0.01

July 16, 2015

Contents

This is the Chainer documentation.

CHAPTER 1

Chainer Tutorial

1.1 Introduction to Chainer

This is the first section of the Chainer Tutorial. In this section, you will learn about the following things:

- Pros and cons of existing frameworks and why we are developing Chainer
- Simple example of forward and backward computation
- Usage of parameterized functions and their gradient computation
- Management of a set of parameterized functions (a.k.a. "model" in most frameworks)
- Parameter optimization

After reading this section, you will be able to:

- Compute gradients of some arithmetics
- Write a multi-layer perceptron with Chainer

1.1.1 Core Concept

As mentioned on the front 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 minibatches. 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. Torch7 and Theano-based frameworks).

In contrast, Chainer adopts a **"Define-by-Run"** scheme, i.e., the network is defined on-the-fly via the actual forward computation. More precisely, Chainer stores the history of computation instead of programming logic. This strategy enables 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.

Note: In example codes of this tutorial, we assume for simplicity that the following symbols are already imported:

```
import numpy as np
from chainer import cuda, Function, FunctionSet, gradient_check, Variable, optimizers
import chainer.functions as F
```

These imports appear widely in Chainer's codes and examples. For simplicity, we omit this idiom in this tutorial.

1.1.2 Forward/Backward Computation

As described above, Chainer uses "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 `Variable` object. Here we start with simple `ndarray` with only one element:

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

Warning: Chainer currently only supports 32-bit float for most computations.

A Variable object has 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 Variable object, whose value can be extracted by accessing the `data` attribute:

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

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 differentiation. 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 are easily generalized to multi-element array input. Note that if we want to start backward computation from a variable holding a multi-element array, we must set the *initial error* manually. This is simply done 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 `functions` module. You can combine them to realize complicated functions with automatic backward computation.

1.1.3 Parameterized functions

In order to write neural networks, we have to use some *parameterized functions* and optimize their parameters. As noted above, functions are predefined in `functions` module, which also includes parameterized functions.

One of the most fundamental parameterized functions is the `Linear` function (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. A linear function from three-dimensional space to two-dimensional space is defined by:

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

Note: Most functions only accept minibatch input, where the first dimension of input arrays is considered as the *batch dimension*. In the above Linear function case, input must has shape of (N, 3), where N is the minibatch size.

The parameters of Linear function are stored in `W` and `b` attributes. By default, the matrix W is initialized randomly, while the vector b is initialized with zeros.

```
>>> f.W
array([[ 1.33545339, -0.01839679,  0.7662735 ],
       [-1.21562171, -0.44784674, -0.07128379]], dtype=float32)
>>> f.b
array([ 0.,  0.], dtype=float32)
```

Instances of a parameterized function class act like usual functions:

```
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = f(x)
>>> y.data
array([[ 3.5974803 , -2.3251667 ],
       [ 9.84747124, -7.52942371]], dtype=float32)
```

Gradients of parameters are computed by `backward()` method. Note that gradients are **accumulated** by the method rather than overwritten. So first you must initialize gradients to zero to renew the computation. Gradients of Linear function are stored in `gW` and `gb` attributes:

```
>>> f.gW.fill(0)
>>> f.gb.fill(0)
```

Note: This procedure is simplified by FunctionSet and Optimizer, which we will see in the next seciton.

Now we can compute the gradients of parameters by simply calling backward method:

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

```
>>> f.gb
array([ 2.,  2.], dtype=float32)
```

1.1.4 FunctionSet

Most neural network architectures contain multiple parameterized functions. `FunctionSet` makes it easy to manage them. This class acts like a simple object, with attributes initialized by keyword arguments of the initializer:

```
>>> model = FunctionSet(
...     l1 = F.Linear(4, 3),
...     l2 = F.Linear(3, 2),
... )
>>> model.l1
<chainer.functions.linear.Linear object at 0x7f7f03e4f350>
>>> model.l2
<chainer.functions.linear.Linear object at 0x7f7f03e4f590>
```

You can also add additional functions later by setting attributes:

```
>>> model.l3 = F.Linear(2, 2)
```

Since the `model` is just an object with functions stored as its attributes, we can use these functions in forward computation:

```
>>> x = Variable(np.array([[1, 2, 3, 4], [5, 6, 7, 8]], dtype=np.float32))
>>> h1 = model.l1(x)
>>> h2 = model.l2(h1)
>>> h3 = model.l3(h2)
```

One of the features of FunctionSet is the ability to collect parameters and gradients. A tuple of all parameters and a tuple of all gradients are extracted by `FunctionSet.parameters` and `FunctionSet.gradients` properties, respectively.

1.1.5 Optimizer

`Optimizer` is the last core feature of Chainer described in this section. It runs a numerical optimization algorithm given tuples of parameters and gradients. Many algorithms are implemented in `optimizers` module. Here we use the simplest one, called Stochastic Gradient Descent:

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

The method `setup()` prepares for the optimization given parameters and gradients. The interface is designed to match the return values of the `FunctionSet.collect_parameters()` method.

Note: Since Optimizer does not know the functions that actually own the parameters and gradients, once parameters and gradients are given to Optimizer, functions must use same parameter and gradient array objects throughout all forward/backward computations.

In order to run optimization, you first have to compute gradients. Zeroing the initial gradient arrays are simply done by calling `zero_grads()` method:

```
>>> optimizer.zero_grads()
```

We have done the zeroing manually in the previous section. The line above is an equivalent and simpler way to initialize the gradients.

Then, after computing gradient of each parameter, `update()` method runs one iteration of optimization:

```
>>> (compute gradient)
>>> optimizer.update()
```

Optimizer also contains some features related to parameter and gradient manipulation, e.g. weight decay and gradient clipping.

1.1.6 Example: Multi-layer Perceptron on MNIST

Now you can solve a multiclass classification task using a multi-layer perceptron. Here we use hand-written digits dataset called MNIST, which is the long-standing de-facto "hello world" of machine learning. This MNIST example is also found in `examples/mnist` directory of the official repository.

In order to use MNIST, `sklearn.datasets.fetch_mldata()` function of scikit-learn is useful:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
```

The mnist dataset consists of 70,000 grayscale images of size 28x28 (i.e. 784 pixels) and corresponding digit labels. First, we scale pixels to [0, 1] values, and divide the dataset into 60,000 training samples and 10,000 test samples.

```
>>> x_all = mnist.data.astype(np.float32) / 255
>>> y_all = mnist.target.astype(np.int32)
>>> x_train, x_test = np.split(x_all, [60000])
>>> y_train, y_test = np.split(y_all, [60000])
```

Next, we want to define the architecture. We use a simple three-layer rectifier network with 100 units per layer as an example. Before defining the forward routine, we have to prepare our parameterized functions:

```
>>> model = FunctionSet(
...     l1 = F.Linear(784, 100),
...     l2 = F.Linear(100, 100),
...     l3 = F.Linear(100,  10),
... )
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model.collect_parameters())
```

Note that `model.l3` is the final linear layer whose output corresponds to the ten digits. We also set up the optimizer here.

Now we can define the forward routine using these Linear functions. Typically it is defined as a simple python function given input arrays:

```
>>> def forward(x_data, y_data):
...     x = Variable(x_data)
...     t = Variable(y_data)
...     h1 = F.relu(model.l1(x))
...     h2 = F.relu(model.l2(h1))
...     y = model.l3(h2)
...     return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

This function uses `functions.relu()` as an activation function. Since ReLU does not have parameters to optimize, it does not need to be included in *model*. `functions.softmax_cross_entropy()` computes the loss function of softmax regression. `functions.accuracy()` computes the classification accuracy of this minibatch.

Finally, we can write a learning loop as following:

```
>>> batchsize = 100
>>> for epoch in xrange(20):
...     print 'epoch', epoch
...     indexes = np.random.permutation(60000)
...     for i in xrange(0, 60000, batchsize):
...         x_batch = x_train[indexes[i : i + batchsize]]
...         y_batch = y_train[indexes[i : i + batchsize]]
...
...         optimizer.zero_grads()
...         loss, accuracy = forward(x_batch, y_batch)
...         loss.backward()
...         optimizer.update()
```

Only the last four lines are the code related to Chainer, which are already described above.

Here you find that, at each iteration, the network is defined by forward computation, used for backprop, and then disposed. By leveraging this "Define-by-Run" scheme, you can imagine that recurrent nets with variable length input are simply handled by just using loop over different length input for each iteration.

After or during optimization, we want to evaluate the model on the test set. It can be achieved simply by calling forward function:

```
>>> sum_loss, sum_accuracy = 0, 0
>>> for i in xrange(0, 10000, batchsize):
...     x_batch = x_test[i : i + batchsize]
...     y_batch = y_test[i : i + batchsize]
...     loss, accuracy = forward(x_batch, y_batch)
...     sum_loss      += loss.data * batchsize
...     sum_accuracy  += accuracy.data * batchsize
...
>>> mean_loss     = sum_loss / 10000
>>> mean_accuracy = sum_accuracy / 10000
```

The example code contains GPU support, though the essential part is same as the code in this tutorial. We will review in later sections how to use GPU(s).

1.2 Recurrent Nets and their Computational Graph

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 volatile variables to prevent network construction

1.2.1 Recurrent Nets

Recurrent nets are neural networks with loops. They are often used to learn from sequential input/output. Given an input stream $x_1, x_2, \ldots, x_t, \ldots$ 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 periodically 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 that there are 1,000 different word types, and that we use 100 dimensional real vectors to represent each word (a.k.a. word embedding).

Before writing the forward computation, we have to define parameterized functions:

```
model = FunctionSet(
    embed  = F.EmbedID(1000, 100),
    x_to_h = F.Linear(100,   50),
    h_to_h = F.Linear( 50,   50),
    h_to_y = F.Linear( 50, 1000),
)
optimizer = optimizers.SGD()
optimizer.setup(model.collect_parameters())
```

Here `EmbedID` is a parameterized function class for word embedding. It converts input integers into corresponding fixed-dimensional embedding vectors. Other Linear layers represent the transformation as their names indicate. Here we use 50 hidden units.

Then, we can write down the forward computation. Suppose that the input word sequence is given as a list of integer arrays. The forward computation is simply written with a for loop:

```
def forward_one_step(h, cur_word, next_word, volatile=False):
    word = Variable(cur_word, volatile=volatile)
    t    = Variable(next_word, volatile=volatile)
    x    = F.tanh(model.embed(word))
    h    = F.tanh(model.x_to_h(x) + model.h_to_h(h))
    y    = model.h_to_y(h)
    loss = F.softmax_cross_entropy(y, t)
    return h, loss

def forward(x_list, volatile=False):
    h = Variable(np.zeros((50,), dtype=np.float32), volatile=volatile)
    loss = 0
    for cur_word, next_word in zip(x_list, x_list[1:]):
        h, new_loss = forward_one_step(h, cur_word, next_word, volatile=volatile)
        loss += new_loss
    return loss
```

We implemented the one-step-forward computation as a separate function, which is a best practice of writing recurrent nets for higher extensibility. Ignore the argument `volatile` for now, we will review it in the next subsection. The `forward` function is very simple and no special care needs to be taken with respect to the length of the input sequence. This code actually handles variable length input sequences without any tricks.

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:

```
optimizer.zero_grads()
loss = forward(x_list)
loss.backward()
optimizer.update()
```

Do not forget to call `Optimizer.zero_grads()` before the backward computation!

1.2.2 Truncate the Graph by Unchaining

Learning from very long sequences is also a typical use case of recurrent nets. Suppose that 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 that 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 `forward_one_step` function that we wrote above.

```
h = Variable(np.zeros((50,), dtype=np.float32))
loss   = 0
count  = 0
seqlen = len(x_list[1:])

for cur_word, next_word in zip(x_list, x_list[1:]):
    h, new_loss = forward_one_step(h, cur_word, next_word)
    loss  += new_loss
    count += 1
    if count % 30 == 0 or count == seqlen:
        optimizer.zero_grads()
        loss.backward()
        loss.unchain_backward()
        optimizer.update()
```

State is updated at `foward_one_step`, 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 latest state `h` itself is not lost, since above code 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.

1.2.3 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 passing `volatile` flag to all input variables. Such variables are called *volatile variables*.

> **Warning:** It is not allowed to mix volatile and non-volatile variables as arguments to same function.

Remember that our `forward` function accepts `volatile` argument. So we can enable volatile forward computation by just passing `volatile=True` to this function:

```
loss = forward(x_list, volatile=True)
```

Volatile variables are also useful to evaluate feed-forward networks.

Variable's volatility can be changed directly by setting the `Variable.volatile` attribute. This enables us to combine a fixed feature extractor network and a trainable predictor network. For example, suppose that we want to train a feed-forward network `predictor_func`, which is located on top of another fixed pretrained 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):

```
x = Variable(x_data, volatile=True)
feat = fixed_func(x)
feat.volatile = False
y = predictor_func(feat)
y.backward()
```

At first, the input variable `x` is volatile, so `fixed_func` is executed in volatile mode, i.e. without memorizing the computation history. Then the intermediate variable `feat` is manually set to non-volatile, so `predictor_func` is executed in non-volatile 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.

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.

1.3 Using GPU(s) in Chainer

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

- Relationship between Chainer and PyCUDA
- Basics of GPUArray
- 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

1.3.1 Relationship between Chainer and PyCUDA

Chainer uses PyCUDA as its backend for GPU computation and the `pycuda.gpuarray.GPUArray` class as the GPU array implementation. GPUArray has far less features compared to `numpy.ndarray`, though it is still enough to implement the required features for Chainer.

Note: `chainer.cuda` module imports many important symbols from PyCUDA. For example, the GPUArray class is referred as `cuda.GPUArray` in the Chainer code.

Chainer provides wrappers of many PyCUDA functions and classes, mainly in order to support customized default allocation mechanism. 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. `cuMemAlloc` and `cuMemFree` functions) synchronize CPU and GPU computations, which hurts performance. In order to avoid memory allocation and deallocation during the computation, Chainer uses PyCUDA's memory pool utilities as the standard memory allocator. Since memory pool is not the default allocator in PyCUDA, Chainer provides many wrapper functions and classes to use memory pools in a simple way. At the same time, Chainer's wrapper functions and classes make it easy to handle multiple GPUs.

Note: Chainer also uses scikit-cuda for a wrapper of CUBLAS, and some functions use CuDNN v2 if available. We omit their usage in this tutorial.

Note: We also do not touch the detail of PyCUDA. See PyCUDA's documentation instead.

1.3.2 Basics of GPUArray in Chainer

In order to use GPU in Chainer, we must initialize `chainer.cuda` module before any GPU-related operations:

```
cuda.init()
```

The `cuda.init()` function initializes global state and PyCUDA. This function accepts an optional argument `device`, which indicates the GPU device ID to select initially.

Warning: If you are using `multiprocessing`, the initialization must take place for each process *after* the fork. The main process is no exception, i.e., `cuda.init()` should not be called before all the children that use GPU have been forked.

Then we can create a GPUArray object using functions of the `cuda` module. Chainer provides many constructor functions resembling the ones of NumPy: `empty()`, `empty_like()`, `full()`, `full_like()`, `zeros()`, `zeros_like()`, `ones()`, `ones_like()`.

Another useful function to create a GPUArray object is `to_gpu()`. This function copies a `numpy.ndarray` object to a newly allocated GPUArray object. For example, the following code

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

generates the same `x_gpu` as the following code:

```
x_gpu = cuda.ones((5, 4, 3))
```

Note: Allocation functions of the `cuda` module use `numpy.float32` as the default element type.

The `cuda` module also has `to_cpu()` function to copy a GPUArray object to an ndarray object:

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

All GPUArray constructors allocate memory on the current device. In order to allocate memory on a different device, we can use device switching utilities. `cuda.use_device()` function changes the current device:

```
cuda.use_device(1)
x_gpu1 = cuda.empty((4, 3))
```

There are many situations in which we want to temporarily switch the device, where the `cuda.using_device()` function is useful. It returns an resource object that can be combinated with the `with` statement:

```
with cuda.using_device(1):
    x_gpu1 = cuda.empty((4, 3))
```

These device switching utilities also accepts a GPUArray object as a device specifier. In this case, Chainer switches the current device to one that the array is allocated on:

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

Warning: An array that is not allocated by Chainer's allocator cannot be used as a device specifier.

A GPUArray object allocated by Chainer can be copied between GPUs by `cuda.copy()` function:

```
cuda.use_device(0)
x0 = cuda.ones((4, 3))
x1 = cuda.copy(x0, out_device=1)
```

1.3.3 Run Neural Networks on a Single GPU

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

A `FunctionSet` object can be transferred to the specified GPU using the `to_gpu()` method. Make sure to give parameters and gradients of the GPU version to the optimizer.

```
model = FunctionSet(
    l1 = F.Linear(784, 100),
    l2 = F.Linear(100, 100),
    l3 = F.Linear(100,  10),
).to_gpu()

optimizer = optimizers.SGD()
optimizer.setup(model.collect_parameters())
```

Note that this method returns the function set itself. The device specifier can be omitted, in which case it uses the current device.

Then, all we have to do is transferring each minibatch to the GPU:

```
batchsize = 100
for epoch in xrange(20):
    print 'epoch', epoch
    indexes = np.random.permutation(60000)
    for i in xrange(0, 60000, batchsize):
        x_batch = cuda.to_gpu(x_train[indexes[i : i + batchsize]])
        y_batch = cuda.to_gpu(y_train[indexes[i : i + batchsize]])

        optimizer.zero_grads()
        loss, accuracy = forward(x_batch, y_batch)
        loss.backward()
        optimizer.update()
```

This is almost identical to the code of the original example, we just inserted a call to the `cuda.to_gpu()` function to the minibatch arrays.

1.3.4 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 first have to define a `FunctionSet`. Be careful that parameters that will be used on a device must reside on that device. Here is a simple example of the model definition:

```
model = FunctionSet(
    gpu0 = FunctionSet(
        l1=F.Linear( 784, 1000),
        l2=F.Linear(1000, 1000),
        l3=F.Linear(1000, 2000),
        l4=F.Linear(2000, 1000),
        l5=F.Linear(1000, 1000),
        l6=F.Linear(1000,   10)
    ).to_gpu(0),
    gpu1 = FunctionSet(
        l1=F.Linear( 784, 1000),
        l2=F.Linear(1000, 1000),
        l3=F.Linear(1000, 2000),
        l4=F.Linear(2000, 1000),
        l5=F.Linear(1000, 1000),
        l6=F.Linear(1000,   10)
    ).to_gpu(1)
)
```

Recall that `FunctionSet.to_gpu()` returns the FunctionSet object itself. Note that FunctionSet can be nested as above.

Now we can define the network architecture that we have shown in the diagram:

```
def forward(x_data, y_data):
    x_0 = Variable(cuda.to_gpu(x_data, 0))
    x_1 = Variable(cuda.to_gpu(x_data, 1))
    t   = Variable(cuda.to_gpu(y_data, 0))

    h1_0 = F.relu(model.gpu0.l1(x_0))
    h1_1 = F.relu(model.gpu1.l1(x_1))

    h2_0 = F.relu(model.gpu0.l2(h1_0))
    h2_1 = F.relu(model.gpu1.l2(h1_1))

    h3_0 = F.relu(model.gpu0.l3(h2_0))
    h3_1 = F.relu(model.gpu1.l3(h2_1))

    # Synchronize
    h3_0 += F.copy(h3_1, 0)
    h3_1  = F.copy(h3_0, 1)
```

```
    h4_0 = F.relu(model.gpu0.l4(h3_0))
    h4_1 = F.relu(model.gpu1.l4(h3_1))

    h5_0 = F.relu(model.gpu0.l5(h4_0))
    h5_1 = F.relu(model.gpu1.l5(h4_1))

    h6_0 = F.relu(model.gpu0.l6(h5_0))
    h6_1 = F.relu(model.gpu1.l6(h5_1))

    # Synchronize
    y = h6_0 + F.copy(h6_1, 0)
    return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

First, recall that `cuda.to_gpu()` accepts an optional argument to specify the device identifier. We use this to transfer the input minibatch to both the 0th and the 1st devices. Then, we can write this model-parallel example employing the `functions.copy()` function. This function transfers an input array to another device. Since it is a function on `Variable`, the operation supports backprop, which 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 most of the GPU computation is asynchronous to the host CPU.

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

1.3.5 Data-parallel Computation on Multiple GPUs

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 the model:

```
model = FunctionSet(
    l1 = F.Linear(784, 100),
    l2 = F.Linear(100, 100),
    l3 = F.Linear(100,  10),
)
```

We have to copy this model into two different devices. This is done by using `copy.deepcopy()` and `FunctionSet.to_gpu()` method:

```
import copy
model_0 = copy.deepcopy(model).to_gpu(0)
model_1 = model.to_gpu(1)
```

Then, set up optimizer as:

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

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`.

Forward function is almost same as the original example:

```
def forward(x_data, y_data, model):
    x = Variable(x_data)
    t = Variable(y_data)
    h1 = F.relu(model.l1(x))
    h2 = F.relu(model.l2(h1))
    y = model.l3(h2)
    return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

The only difference is that `forward` accepts `model` as an argument. We can feed it with a model and arrays on an appropriate device. Then, we can write a data-parallel learning loop as follows:

```
batchsize = 100
for epoch in xrange(20):
    print 'epoch', epoch
    indexes = np.random.permutation(60000)
    for i in xrange(0, 60000, batchsize):
        x_batch = x_train[indexes[i : i + batchsize]]
        y_batch = y_train[indexes[i : i + batchsize]]

        optimizer.zero_grads()

        loss_0, accuracy_0 = forward(
            cuda.to_gpu(x_batch[:batchsize//2], 0),
            cuda.to_gpu(y_batch[:batchsize//2], 0),
            model_0)
        loss_0.backward()

        loss_1, accuracy_1 = forward(
            cuda.to_gpu(x_batch[batchsize//2:], 1),
            cuda.to_gpu(y_batch[batchsize//2:], 1),
            model_1)
        loss_1.backward()

        optimizer.acumulate_grads(model_1.gradients)
        optimizer.update()

        model_1.copy_parameters_from(model_0.parameters)
```

One half of the minibatch is forwarded to GPU 0, the other half to GPU 1. Then the gradients are accumulated by the `Optimizer.accumulate_grads()` method. 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 `FunctionSet.copy_parameters_from()` method.

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.

1.4 Define your own function

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

- How to define a non-parameterized function
- Useful tools to write a function using a GPU

- How to define a parameterized function
- How to test the function definition

After reading this section, you will be able to:

- Write your own non-parameterized function
- Define simple kernels in the function definition
- Write your own parameterized function

1.4.1 Non-parameterized Functions

Chainer provides a collection of functions in the *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.

Since they are simpler, we first show how to define non-parameterized 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. Any function must inherit the `Function` class. The skeleton of a non-parameterized 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
        return w,

    def backward_cpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw      = grad_outputs[0]

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

As per the warning above, `forward_cpu` function 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 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.).

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[0]

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

In GPU methods, arrays are of type `pycuda.gpuarray.GPUArray` We use arithmetic operators defined for GPUArray. These operators implement the basic elementwise arithmetics.

You maybe 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[0]

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

Note that this is a very rare case, since GPUArray does not implement most features of `numpy.ndarray`.

1.4.2 Write an Elementwise Kernel Function

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 and backward computations, which may hurt performance. We can reduce the number of invocations by defining our own kernel.

Most functions only require elementwise operations like MulAdd. PyCUDA provides a useful tool to define elementwise kernels, the `pycuda.elementwise.ElementwiseKernel` class, and Chainer wraps it by `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):
        x, y, z = inputs
        w = cuda.empty_like(x)
        cuda.elementwise(
            'float* w, const float* x, const float* y, const float* z',
            'w[i] = x[i] * y[i] + z[i]',
            'muladd_fwd')(w, x, y, z)
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw      = grad_outputs[0]

        gx = cuda.empty_like(x)
        gy = cuda.empty_like(y)
        cuda.elementwise(
            '''
               float* gx, float* gy,
               const float* x, const float* y, const float* gw
            ''', '''
               gx[i] = gy[i] * gw[i];
               gy[i] = gx[i] * gw[i];
            ''', 'muladd_bwd')(gx, gy, x, y, gw)

        gz = gw  # no copy
        return gx, gy, gz
```

`cuda.elementwise()` function accepts the essential implentation of the kernel function, and returns a kernel invokation function (actually, it returns `ElementwiseKernel` object, which is callable). In typical usage, we pass three arguments to this function. The first is an argument list of the kernel function. The second is a body of *parallel loop*, where the variable `i` indicates the index in the loop. Note that `i` runs through all indexes of the first array argument by default. The third is the name of the kernel function, which is shown in debugger and profilers.

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

The first one is *binary caching*: `cuda.elementwise()` function caches the compiled binary in the `/tmp` 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 PyCUDA.

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. `cuda.elementwise()` function memoizes the arguments and the curent context, and if it is called with

the same arguments and the same context, it reuses the previously uploaded kernel code.

1.4.3 Parameterized Functions

Next we show how to define a parameterized function. At this time, suppose that we want to implement elementwise product function between the input array and the parameter array.

Note: Note that the elementwise product between a variable and parameters can be simply implemented by `functions.Parameter` function:

```
p = F.Parameter(np.random.rand((4, 3), dtype=np.float32))
x = Variable(...)
y = p() * x
```

The Parameter function takes no arguments and just returns a variable holding the parameter array. The example in this subsection may be slightly more efficient with respect to memory consumption, though.

There are two differences between parameterized functions and non-parameterized functions:

- Parameterized functions have parameter arrays and corresponding gradient arrays. They are typically stored as attributes of the function class, where the function should provide `parameter_names` and `gradient_names` attributes (or properties). Otherwise, the function must override `parameters` and `gradients` properties directly.
- Parameterized functions must accumulate gradients on backward.

Note that gradient arrays are automatically zeroed by an optimizer, so function implementation only need to initialize their shapes. Then, the implementation of elementwise product may be as following:

```
class EltwiseParamProduct(Function):
    parameter_names = 'w',
    gradient_names  = 'gw',

    def __init__(self, shape):
        self.w  = np.random.randn(shape).astype(np.float32)
        self.gw = np.empty_like(self.w)

    def forward(self, inputs):
        x = inputs[0]
        y = self.w * x
        return y,

    def backward(self, inputs, grad_outputs):
        x  = inputs[0]
        gy = grad_outputs[0]

        self.gw += gy * x
        gx       = gy * self.w

        return gx,
```

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. It does not make any trouble even if the function object is used more than once in the same network, since `Function.__call__()` operator copies itself before the forward computation.

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.

1.4.4 Testing Function

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 `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 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:

```
class TestReLU(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()

        func = y.creator
        f = lambda: func.forward((x.data,))
        gx, = gradient_check.numerical_grad(f, (x.data,), (y.grad,))

        gradient_check.assert_allclose(gx, x.grad)
```

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

You can find many examples of function tests under `tests/function_tests` directory.

CHAPTER 2

Chainer Reference Manual

2.1 Core functionalities

2.1.1 Variable

2.1.2 Function

2.1.3 FunctionSet

2.1.4 Optimizer

2.2 Utilities

2.2.1 CUDA utilities

Initialization and global states

Devices and contexts

GPUArray allocation and copy

Random number generators

Kernel definition utilities

Interprocess communication on GPU

2.2.2 Common algorithms

2.3 Assertion and Testing

Chainer provides some facilities to make debugging easy.

`Function` 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.

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.

2.3.1 Type checking utilites

2.3.2 Gradient checking utilities

2.4 Standard Function implementations

Chainer provides basic `Function` implementations in the *`chainer.functions`* package.

Non-parameterized functions are provided as plain Python functions. These can be used directly in forward computation without explicit handling of `Function` objects. On the other hand, parameterized functions should be used with explicit handling of `Function` objects.

2.4.1 Learnable connections

2.4.2 Array commputation functions

2.4.3 Array manipulation functions

2.4.4 Array computations

2.4.5 Activation functions

2.4.6 Pooling functions

2.4.7 Normalization functions

2.4.8 Loss, evaluation and aggregation

2.4.9 Reusable subnetwork of complex architectures

2.5 Optimizers

2.6 Caffe Reference 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, one of the reason why this framework gets popular.

Chainer can import the reference models and emulate the network by `Function` implementations. This functionality is provided by the `chainer.functions.caffe.CaffeFunction` class.

2.7 Visualization of Computational Graph

As neural networks get larger and complicated, it gets much harger 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`.

CHAPTER 3

Chainer Contribution Guide

This is a guide for all contributions to Chainer. The development of Chainer is running on the official repository at GitHub. Anyone that wants to register an issue or to send a pull request should read through this document.

3.1 Classification of Contributions

There are several ways to contribute to Chainer community:

1. Registering an issue
2. Sending a pull request (PR)
3. Sending a question to Chainer User Group
4. Open-sourcing an external example
5. Writing a post about Chainer

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

3.2 Release and Milestone

We are using GitHub Flow as our basic working process. In particular, we are using the master branch for our development, and releases are made as tags.

Releases are classified into three groups: major, minor, and revision. This classification is based on following criteria:

- A **major** release contains catastrophic changes on the interface that may break existing user codes.
- A **minor** release contains additions and modifications on the interface. It may break some existing user codes, though they must be fixed by small efforts.
- A **revision** release contains changes that does not affect the documented interface. It mainly consists of bug fixes, implementation improvements, and test/document/example updates.

The release classification is reflected into the version number x.y.z, where x, y, and z corresponds to major, minor, and revision updates, respectively.

We sets milestones for some future releases. A milestone for a revision release is set right after the last release. On the other hand, a milestone for a minor or major release is set four weeks prior to its due.

3.3 Issues and PRs

Issues and PRs are classified into following categories:

- **Bug**: bug reports (isseus) and bug fixes (PRs)
- **Enhancement**: implementation improvements without breaking the interface
- **Feature**: feature requests (issues) and their implementations (PRs)
- **Test**: test fixes and updates
- **Document**: document fixes and improvements
- **Example**: fixes and improvements on the examples
- **Other**: other issues and PRs

Issues and PRs are labeled by these categories. This classification is often reflected into its corresponding release category: Feature issues/PRs are contained into minor/major releases, while other issues/PRs can be contained into any releases including revision ones.

On registering an issue, write precise explanations on what you want Chainer to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include **what** you want to do (and **why** you want to do, if needed). You can contain your thoughts on **how** to realize it into the feature requests, though **what** part is most important for discussions.

> **Warning:** If you have a question on usages of Chainer, it is highly recommended to send a post to Chainer User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may redirect question issues to Chainer User Group.

If you can write codes to fix an issue, send a PR to the master branch. Before writing your codes for PRs, read through the *Coding Guidelines*. The description of any PR must contain a precise explanation of **what** and **how** you want to do; it is the first documentation of your codes for developers, a very important part of your PR.

Once you send a PR, it is automatically tested on Travis CI. After the automatic test passes, some of the core developers will start reviewing your codes. Note that this automatic PR test only includes CPU tests.

Note: We are also running continuous integrations with GPU tests for the master branch. Since this service is running on our internal server, we do not use it for automatic PR tests to keep the server secure.

Even if your codes are not complete, you can send a pull request as a *work-in-progress PR* by putting the `[WIP]` prefix to the PR title. If you write a precise explanation about the PR, core developers and other contributors can join the discussion about how to proceed the PR.

3.4 Coding Guidelines

We use PEP8 and a part of OpenStack Style Guidelines related to general coding style as our basic style guidelines.

Before checking your code, you can use automatic formatter to set appropriate spacing, etc. We recommend you to install the `pyformat` and `isort` packages, and run the following commands:

```
$ pyformat -i path/to/your/code.py
$ isort path/to/your/code.py
```

Note that these formatters do not cover all part of the style guidelines.

To check your code, use `flake8` command installed by `hacking` package:

```
$ pip install hacking
$ flake8 path/to/your/code.py
```

The `flake8` command lets you know the part of your code not obeying our style guidelines. Before sending a pull request, be sure to check that your code passes the `flake8` checking.

Note that `flake8` command is not perfect. It does not check some of the style guidelines. Here is a (not-complete) 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 symbols* in our code base. They are symbols imported by packages and subpackages of `chainer`. For example, `chainer.Variable` is a shortcut of `chainer.variable.Variable`. **It is not allowed to use such shortcuts in the ‘‘chainer‘‘ library implementation**. Note that you can still use them in `tests` and `examples` directories.

Once you send a pull request, your coding style is automatically checked by Travis-CI. The reviewing process starts after the check passes.

3.5 Testing Guidelines

Testing is one of the most important part of your code. You must test your code by unit tests following our testing guidelines.

We are using `nose` package to run unit tests. You can run unit tests simply by running `nosetests` command under the repository root. It requires CUDA by default. In order to run unit tests that do not require CUDA, pass `--attr='!gpu'` option to the nosetests command:

```
$ nosetests path/to/your/test.py --attr='!gpu'
```

Tests are put into the `tests` directory. This directory has the same structure as the `chainer` directory. In order to enable test runner to find test scripts correctly, we are using 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.

Following this naming convention, you can run all the tests by just typing `nosetests` at the repository root:

```
$ nosetests
```

If you modify the code related to existing unit tests, you must run this command.

There are many examples of unit tests under the `tests` directory. They simply use the `unittest` package of the standard library.

If your patch includes GPU-related code, your tests must run with and without GPU capability. Test functions that requires CUDA must be tagged by 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 by the `chainer.testing.attr.gpu` decorator are skipped if `--attr='!gpu'` is given. We also have the `chainer.testing.attr.cudnn` decorator to let nosetests know that the test depends on CuDNN.

Once you send a pull request, your code is automatically tested by Travis-CI **with –attr='!gpu' option**. Since Travis-CI does not support CUDA, we cannot check you CUDA-related code automatically. The reviewing process starts after the test passes. Note that reviewers will test your code without the option to check CUDA-related code.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

C

C