Blocks Documentation

Release 0.2.0

Université de Montréal

Sep 06, 2018

Contents

Blocks is a framework that helps you build and manage neural network models on using Theano.

Want to get try it out? Start by *installing* Blocks and having a look at the *quickstart* further down this page. Once you're hooked, try your hand at the *tutorials* and the examples.

Blocks is developed in parallel with Fuel, a dataset processing framework.

Warning: Blocks is a new project which is still under development. As such, certain (all) parts of the framework are subject to change. The last stable (and thus likely an outdated) version can be found in the `stable` branch.

Tip: That said, if you are interested in using Blocks and run into any problems, feel free to ask your question on the mailing list. Also, don't hesitate to file bug reports and feature requests by making a GitHub issue.

CHAPTER 1

Tutorials

1.1 Installation

The easiest way to install Blocks is using the Python package manager `pip`. Blocks isn't listed yet on the Python Package Index (PyPI), so you will have to grab it directly from GitHub.

```
$ pip install git+git://github.com/mila-udem/blocks.git \
  -r https://raw.githubusercontent.com/mila-udem/blocks/master/requirements.txt
```

This will give you the cutting-edge development version. The latest stable release is in the `stable` branch and can be installed as follows.

```
$ pip install git+git://github.com/mila-udem/blocks.git@stable \
  -r https://raw.githubusercontent.com/mila-udem/blocks/stable/requirements.txt
```

Note: Blocks relies on several packages, such as Theano and picklable_itertools, to be installed directly from GitHub. The only way of doing so reliably is through a `requirements.txt` file, which is why this installation command might look slightly different from what you're used to.

Installing requirements from GitHub requires pip 1.5 or higher; you can update with `pip update pip`.

If you don't have administrative rights, add the `--user` switch to the install commands to install the packages in your home folder. If you want to update Blocks, simply repeat the first command with the `--upgrade` switch added to pull the latest version from GitHub.

Warning: Pip may try to install or update NumPy and SciPy if they are not present or outdated. However, pip's versions might not be linked to an optimized BLAS implementation. To prevent this from happening make sure you update NumPy and SciPy using your system's package manager (e.g. `apt-get` or `yum`), or use a Python distribution like Anaconda, before installing Blocks. You can also pass the `--no-deps` switch and install all the requirements manually.

If the installation crashes with `ImportError: No module named numpy.distutils.core`, install NumPy and try again again.

1.1.1 Requirements

Blocks' requirements are

- Theano, for pretty much everything
- PyYAML, to parse the configuration file
- six, to support both Python 2 and 3 with a single codebase
- Toolz, to add a bit of functional programming where it is needed

Bokeh is an optional requirement for if you want to use live plotting of your training progress (part of `blocks-extras_`).

nose2 is an optional requirement, used to run the tests.

We develop using the bleeding-edge version of Theano, so be sure to follow the relevant installation instructions to make sure that your Theano version is up to date if you didn't install it through Blocks.

1.1.2 Development

If you want to work on Blocks' development, your first step is to fork Blocks on GitHub. You will now want to install your fork of Blocks in editable mode. To install in your home directory, use the following command, replacing `USER` with your own GitHub user name:

```
$ pip install -e git+git@github.com:USER/blocks.git#egg=blocks[test,docs] --src=$HOME␣
↪\
  -r https://raw.githubusercontent.com/mila-udem/blocks/master/requirements.txt
```

As with the usual installation, you can use `--user` or `--no-deps` if you need to. You can now make changes in the `blocks` directory created by pip, push to your repository and make a pull request.

If you had already cloned the GitHub repository, you can use the following command from the folder you cloned Blocks to:

```
$ pip install -e file:.#egg=blocks[test,docs] -r requirements.txt
```

Documentation

If you want to build a local copy of the documentation, follow the instructions at the *documentation development guidelines*.

1.2 Introduction tutorial

In this tutorial we will perform handwriting recognition by training a multilayer perceptron (MLP) on the MNIST handwritten digit database.

1.2.1 The Task

MNIST is a dataset which consists of 70,000 handwritten digits. Each digit is a grayscale image of 28 by 28 pixels. Our task is to classify each of the images into one of the 10 categories representing the numbers from 0 to 9.

Fig. 1: Sample MNIST digits

1.2.2 The Model

We will train a simple MLP with a single hidden layer that uses the rectifier activation function. Our output layer will consist of a softmax function with 10 units; one for each class. Mathematically speaking, our model is parametrized by θ, defined as the weight matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$, and bias vectors $\mathbf{b}^{(1)}$ and $\mathbf{b}^{(2)}$. The rectifier activation function is defined as

$$\mathrm{ReLU}(\mathbf{x})_i = \max(0, \mathbf{x}_i)$$

and our softmax output function is defined as

$$\mathrm{softmax}(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{n} e^{\mathbf{x}_j}}$$

Hence, our complete model is

$$f(\mathbf{x}; \theta) = \mathrm{softmax}(\mathbf{W}^{(2)}\mathrm{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)})$$

Since the output of a softmax sums to 1, we can interpret it as a categorical probability distribution: $f(\mathbf{x})_c = \hat{p}(y = c \mid \mathbf{x})$, where $\mathbf{x}$ is the 784-dimensional (28 × 28) input and $c \in \{0, ..., 9\}$ one of the 10 classes. We can train the parameters of our model by minimizing the negative log-likelihood i.e. the cross-entropy between our model's output and the target distribution. This means we will minimize the sum of

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_{c=0}^{9} \mathbf{1}_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$

(where $\mathbf{1}$ is the indicator function) over all examples. We use stochastic gradient descent (SGD) on mini-batches for this.

1.2.3 Building the model

Blocks uses "bricks" to build models. Bricks are **parametrized Theano operations**. You can read more about it in the *building with bricks* tutorial.

Constructing the model with Blocks is very simple. We start by defining the input variable using Theano.

Tip: Want to follow along with the Python code? If you are using IPython, enable the doctest mode using the special `%doctest_mode` command so that you can copy-paste the examples below (including the >>> prompts) straight into the IPython interpreter.

```
>>> from theano import tensor
>>> x = tensor.matrix('features')
```

Note that we picked the name `'features'` for our input. This is important, because the name needs to match the name of the data source we want to train on. MNIST defines two data sources: `'features'` and `'targets'`.

For the sake of this tutorial, we will go through building an MLP the long way. For a much quicker way, skip right to the end of the next section. We begin with applying the linear transformations and activations.

We start by initializing bricks with certain parameters e.g. `input_dim`. After initialization we can apply our bricks on Theano variables to build the model we want. We'll talk more about bricks in the next tutorial, *Building with bricks*.

```
>>> from blocks.bricks import Linear, Rectifier, Softmax
>>> input_to_hidden = Linear(name='input_to_hidden', input_dim=784, output_dim=100)
>>> h = Rectifier().apply(input_to_hidden.apply(x))
>>> hidden_to_output = Linear(name='hidden_to_output', input_dim=100, output_dim=10)
>>> y_hat = Softmax().apply(hidden_to_output.apply(h))
```

1.2.4 Loss function and regularization

Now that we have built our model, let's define the cost to minimize. For this, we will need the Theano variable representing the target labels.

```
>>> y = tensor.lmatrix('targets')
>>> from blocks.bricks.cost import CategoricalCrossEntropy
>>> cost = CategoricalCrossEntropy().apply(y.flatten(), y_hat)
```

To reduce the risk of overfitting, we can penalize excessive values of the parameters by adding a $L2$-regularization term (also known as *weight decay*) to the objective function:

$$l(\mathbf{f}(\mathbf{x}), y) = -\log f(\mathbf{x})_y + \lambda_1 \|\mathbf{W}^{(1)}\|^2 + \lambda_2 \|\mathbf{W}^{(2)}\|^2$$

To get the weights from our model, we will use Blocks' annotation features (read more about them in the *Managing the computation graph* tutorial).

```
>>> from blocks.roles import WEIGHT
>>> from blocks.graph import ComputationGraph
>>> from blocks.filter import VariableFilter
>>> cg = ComputationGraph(cost)
>>> W1, W2 = VariableFilter(roles=[WEIGHT])(cg.variables)
>>> cost = cost + 0.005 * (W1 ** 2).sum() + 0.005 * (W2 ** 2).sum()
>>> cost.name = 'cost_with_regularization'
```

Note: Note that we explicitly gave our variable a name. We do this so that when we monitor the performance of our model, the progress monitor will know what name to report in the logs.

Here we set $\lambda_1 = \lambda_2 = 0.005$. And that's it! We now have the final objective function we want to optimize.

But creating a simple MLP this way is rather cumbersome. In practice, we would have used the *MLP* class instead.

```
>>> from blocks.bricks import MLP
>>> mlp = MLP(activations=[Rectifier(), Softmax()], dims=[784, 100, 10]).apply(x)
```

1.2.5 Initializing the parameters

When we constructed the *Linear* bricks to build our model, they automatically allocated Theano shared variables to store their parameters in. All of these parameters were initially set to NaN. Before we start training our network, we will want to initialize these parameters by sampling them from a particular probability distribution. Bricks can do this for you.

```
>>> from blocks.initialization import IsotropicGaussian, Constant
>>> input_to_hidden.weights_init = hidden_to_output.weights_init =␣
↪IsotropicGaussian(0.01)
>>> input_to_hidden.biases_init = hidden_to_output.biases_init = Constant(0)
>>> input_to_hidden.initialize()
>>> hidden_to_output.initialize()
```

We have now initialized our weight matrices with entries drawn from a normal distribution with a standard deviation of 0.01.

```
>>> W1.get_value()
        array([[ 0.01624345, -0.00611756, -0.00528172, ...,  0.00043597, ...
```

1.2.6 Training your model

Besides helping you build models, Blocks also provides the main other features needed to train a model. It has a set of training algorithms (like SGD), an interface to datasets, and a training loop that allows you to monitor and control the training process.

We want to train our model on the training set of MNIST. We load the data using the Fuel framework. Have a look at this tutorial to get started.

After having configured Fuel, you can load the dataset.

```
>>> from fuel.datasets import MNIST
>>> mnist = MNIST(("train",))
```

Datasets only provide an interface to the data. For actual training, we will need to iterate over the data in minibatches. This is done by initiating a data stream which makes use of a particular iteration scheme. We will use an iteration scheme that iterates over our MNIST examples sequentially in batches of size 256.

```
>>> from fuel.streams import DataStream
>>> from fuel.schemes import SequentialScheme
>>> from fuel.transformers import Flatten
>>> data_stream = Flatten(DataStream.default_stream(
...     mnist,
...     iteration_scheme=SequentialScheme(mnist.num_examples, batch_size=256)))
```

The training algorithm we will use is straightforward SGD with a fixed learning rate.

```
>>> from blocks.algorithms import GradientDescent, Scale
>>> algorithm = GradientDescent(cost=cost, parameters=cg.parameters,
...                             step_rule=Scale(learning_rate=0.1))
```

During training we will want to monitor the performance of our model on a separate set of examples. Let's create a new data stream for that.

```
>>> mnist_test = MNIST(("test",))
>>> data_stream_test = Flatten(DataStream.default_stream(
...     mnist_test,
...     iteration_scheme=SequentialScheme(
...         mnist_test.num_examples, batch_size=1024)))
```

In order to monitor our performance on this data stream during training, we need to use one of Blocks' extensions, namely the *DataStreamMonitoring* extension.

```
>>> from blocks.extensions.monitoring import DataStreamMonitoring
>>> monitor = DataStreamMonitoring(
...     variables=[cost], data_stream=data_stream_test, prefix="test")
```

We can now use the *MainLoop* to combine all the different bits and pieces. We use two more extensions to make our training stop after a single epoch and to make sure that our progress is printed.

```
>>> from blocks.main_loop import MainLoop
>>> from blocks.extensions import FinishAfter, Printing
>>> main_loop = MainLoop(data_stream=data_stream, algorithm=algorithm,
...                      extensions=[monitor, FinishAfter(after_n_epochs=1),␣
↪Printing()])
>>> main_loop.run()

--------------------------------------------------------------------------------
BEFORE FIRST EPOCH
--------------------------------------------------------------------------------
Training status:
     epochs_done: 0
     iterations_done: 0
Log records from the iteration 0:
     test_cost_with_regularization: 2.34244632721


--------------------------------------------------------------------------------
AFTER ANOTHER EPOCH
--------------------------------------------------------------------------------
Training status:
     epochs_done: 1
     iterations_done: 235
Log records from the iteration 235:
     test_cost_with_regularization: 0.664899230003
     training_finish_requested: True


--------------------------------------------------------------------------------
TRAINING HAS BEEN FINISHED:
--------------------------------------------------------------------------------
Training status:
     epochs_done: 1
     iterations_done: 235
Log records from the iteration 235:
     test_cost_with_regularization: 0.664899230003
     training_finish_requested: True
     training_finished: True
```

1.3 Building with bricks

Blocks is a framework that is supposed to make it easier to build complicated neural network models on top of Theano. In order to do so, we introduce the concept of "bricks", which you might have already come across in *the introduction tutorial*.

1.3.1 Bricks life-cycle

Blocks uses "bricks" to build models. Bricks are **parametrized Theano operations**. A brick is usually defined by a set of *attributes* and a set of *parameters*, the former specifying the attributes that define the Block (e.g., the number of input and output units), the latter representing the parameters of the brick object that will vary during learning (e.g., the weights and the biases).

The life-cycle of a brick is as follows:

1. **Configuration:** set (part of) the *attributes* of the brick. Can take place when the brick object is created, by setting the arguments of the constructor, or later, by setting the attributes of the brick object. No Theano variable is created in this phase.
2. **Allocation:** (optional) allocate the Theano shared variables for the *parameters* of the Brick. When `allocate()` is called, the required Theano variables are allocated and initialized by default to `NaN`.
3. **Application:** instantiate a part of the Theano computational graph, linking the inputs and the outputs of the brick through its *parameters* and according to the *attributes*. Cannot be performed (i.e., results in an error) if the Brick object is not fully configured.
4. **Initialization:** set the **numerical values** of the Theano variables that store the *parameters* of the Brick. The user-provided value will replace the default initialization value.

Note: If the Theano variables of the brick object have not been allocated when `apply()` is called, Blocks will quietly call `allocate()`.

Example

Bricks take Theano variables as inputs, and provide Theano variables as outputs.

```
>>> import theano
>>> from theano import tensor
>>> from blocks.bricks import Tanh
>>> x = tensor.vector('x')
>>> y = Tanh().apply(x)
>>> print(y)
tanh_apply_output
>>> isinstance(y, theano.Variable)
True
```

This is clearly an artificial example, as this seems like a complicated way of writing `y = tensor.tanh(x)`. To see why Blocks is useful, consider a very common task when building neural networks: Applying a linear transformation (with optional bias) to a vector, and then initializing the weight matrix and bias vector with values drawn from a particular distribution.

```
>>> from blocks.bricks import Linear
>>> from blocks.initialization import IsotropicGaussian, Constant
```

(continues on next page)

(continued from previous page)

```
>>> linear = Linear(input_dim=10, output_dim=5,
...                 weights_init=IsotropicGaussian(),
...                 biases_init=Constant(0.01))
>>> y = linear.apply(x)
```

So what happened here? We constructed a brick called *Linear* with a particular configuration: the input dimension (10) and output dimension (5). When we called *Linear.apply*, the brick automatically constructed the shared Theano variables needed to store its parameters. In the lifecycle of a brick we refer to this as *allocation*.

```
>>> linear.parameters
[W, b]
>>> linear.parameters[1].get_value()
array([ nan,  nan,  nan,  nan,  nan])
```

By default, all our parameters are set to `NaN`. To initialize them, simply call the *initialize()* method. This is the last step in the brick lifecycle: *initialization*.

```
>>> linear.initialize()
>>> linear.parameters[1].get_value()
array([ 0.01,  0.01,  0.01,  0.01,  0.01])
```

Keep in mind that at the end of the day, bricks just help you construct a Theano computational graph, so it is possible to mix in regular Theano statements when building models. (However, you might miss out on some of the niftier features of Blocks, such as variable annotation.)

```
>>> z = tensor.max(y + 4)
```

1.3.2 Lazy initialization

In the example above we configured the *Linear* brick during initialization. We specified input and output dimensions, and specified the way in which weight matrices should be initialized. But consider the following case, which is quite common: We want to take the output of one model, and feed it as an input to another model, but the output and input dimensions don't match, so we will need to add a linear transformation in the middle.

To support this use case, bricks allow for *lazy initialization*, which is turned on by default. This means that you can create a brick without configuring it fully (or at all):

```
>>> linear2 = Linear(output_dim=10)
>>> print(linear2.input_dim)
NoneAllocation
```

Of course, as long as the brick is not configured, we cannot actually apply it!

```
>>> linear2.apply(x)
Traceback (most recent call last):
  ...
ValueError: allocation config not set: input_dim
```

We can now easily configure our brick based on other bricks.

```
>>> linear2.input_dim = linear.output_dim
>>> linear2.apply(x)
linear_apply_output
```

In the examples so far, the allocation of the parameters has always happened implicitly when calling the `apply` methods, but it can also be called explicitly. Consider the following example:

```
>>> linear3 = Linear(input_dim=10, output_dim=5)
>>> linear3.parameters
Traceback (most recent call last):
    ...
AttributeError: 'Linear' object has no attribute 'parameters'
>>> linear3.allocate()
>>> linear3.parameters
[W, b]
```

1.3.3 Nested bricks

Many neural network models, especially more complex ones, can be considered hierarchical structures. Even a simple multi-layer perceptron consists of layers, which in turn consist of a linear transformation followed by a non-linear transformation.

As such, bricks can have *children*. Parent bricks are able to configure their children, to e.g. make sure their configurations are compatible, or have sensible defaults for a particular use case.

```
>>> from blocks.bricks import MLP, Logistic
>>> mlp = MLP(activations=[Logistic(name='sigmoid_0'),
...            Logistic(name='sigmoid_1')], dims=[16, 8, 4],
...            weights_init=IsotropicGaussian(), biases_init=Constant(0.01))
>>> [child.name for child in mlp.children]
['linear_0', 'sigmoid_0', 'linear_1', 'sigmoid_1']
>>> y = mlp.apply(x)
>>> mlp.children[0].input_dim
16
```

We can see that the *MLP* brick automatically constructed two child bricks to perform the linear transformations. When we applied the MLP to `x`, it automatically configured the input and output dimensions of its children. Likewise, when we call *initialize()*, it automatically pushed the weight matrix and biases initialization configuration to its children.

```
>>> mlp.initialize()
>>> mlp.children[0].parameters[0].get_value()
array([[-0.38312393, -1.7718271 ,  0.78074479, -0.74750996],
       ...
       [ 1.32390416, -0.56375355, -0.24268186, -2.06008577]])
```

There are cases where we want to override the way the parent brick configured its children. For example in the case where we want to initialize the weights of the first layer in an MLP slightly differently from the others. In order to do so, we need to have a closer look at the life cycle of a brick. In the first two sections we already talked talked about the three stages in the life cycle of a brick:

1. Construction of the brick
2. Allocation of its parameters
3. Initialization of its parameters

When dealing with children, the life cycle actually becomes a bit more complicated. (The full life cycle is documented as part of the *Brick* class.) Before allocating or initializing parameters, the parent brick calls its *push_allocation_config()* and *push_initialization_config()* methods, which configure the children. If you want to override the child configuration, you will need to call these methods manually, after which you can override the child bricks' configuration.

```
>>> mlp = MLP(activations=[Logistic(name='sigmoid_0'),
...           Logistic(name='sigmoid_1')], dims=[16, 8, 4],
...           weights_init=IsotropicGaussian(), biases_init=Constant(0.01))
>>> y = mlp.apply(x)
>>> mlp.push_initialization_config()
>>> mlp.children[0].weights_init = Constant(0.01)
>>> mlp.initialize()
>>> mlp.children[0].parameters[0].get_value()
array([[ 0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01],
       ...
       [ 0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01]])
```

1.4 Managing the computation graph

Theano constructs computation graphs of mathematical expressions. Bricks help you *build these graphs*, but they do more than that. When you apply a brick to a Theano variable, it automatically *annotates* this Theano variable, in two ways:

- It defines the *role* this variable plays in the computation graph e.g. it will label weight matrices and biases as parameters, keep track of which variables were the in- and outputs of your bricks, and more.
- It constructs *auxiliary variables*. These are variables which are not outputs of your brick, but might still be of interest. For example, if you are training a neural network, you might be interested to know the norm of your weight matrices, so Blocks attaches these as auxiliary variables to the graph.

1.4.1 Using annotations

The *ComputationGraph* class provides an interface to this annotated graph. For example, let's say we want to train an autoencoder using weight decay on some of the layers.

```
>>> from theano import tensor
>>> x = tensor.matrix('features')
>>> from blocks.bricks import MLP, Logistic, Rectifier
>>> from blocks.initialization import IsotropicGaussian, Constant
>>> mlp = MLP(activations=[Rectifier()] * 2 + [Logistic()],
...           dims=[784, 256, 128, 784],
...           weights_init=IsotropicGaussian(), biases_init=Constant(0.01))
>>> y_hat = mlp.apply(x)
>>> from blocks.bricks.cost import BinaryCrossEntropy
>>> cost = BinaryCrossEntropy().apply(x, y_hat)
```

Our Theano computation graph is now defined by our loss, `cost`. We initialize the managed graph.

```
>>> from blocks.graph import ComputationGraph
>>> cg = ComputationGraph(cost)
```

We will find that there are many variables in this graph.

```
>>> print(cg.variables)
[TensorConstant{0}, b, W_norm, b_norm, features, TensorConstant{1.0}, ...]
```

To apply weight decay, we only need the weights matrices. These have been tagged with the *WEIGHT* role. So let's create a filter that finds these for us.

```
>>> from blocks.filter import VariableFilter
>>> from blocks.roles import WEIGHT
>>> print(VariableFilter(roles=[WEIGHT])(cg.variables))
[W, W, W]
```

Note that the variables in *cg.variables* are ordered according to the *topological order* of their apply nodes. This means that for a feedforward network the parameters will be returned in the order of our layers.

But let's imagine for a second that we are actually dealing with a far more complicated network, and we want to apply weight decay to the parameters of one layer in particular. To do that, we can filter the variables by the bricks that created them.

```
>>> second_layer = mlp.linear_transformations[1]
>>> from blocks.roles import PARAMETER
>>> var_filter = VariableFilter(roles=[PARAMETER], bricks=[second_layer])
>>> print(var_filter(cg.variables))
[b, W]
```

Note: There are a variety of different roles that you can filter by. You might have noted already that there is a hierarchy to many of them: Filtering by *PARAMETER* will also return variables of the child roles *WEIGHT* and *BIAS*.

We can also see what auxiliary variables our bricks have created. These might be of interest to monitor during training, for example.

```
>>> print(cg.auxiliary_variables)
[W_norm, b_norm, W_norm, b_norm, W_norm, b_norm]
```

1.5 Live plotting

Note: The live plotting functionality is part of `blocks-extras`, which must be separately installed.

Plots often give a clearer image of your training progress than textual logs. This is why Blocks has a `Plot` extension which allows you to plot the entries from the log that you are interested in.

We use Bokeh, an interactive visualization library, to perform the plotting. More specifically, we use the *Bokeh Plot Server*. This is basically a light web server to which Blocks can send data, which then gets displayed in live plots in your browser. The advantage of this approach is that you can even monitor your models' training progress over a network.

First, make sure that you installed the necessary requirements (see *the installation instructions*). To start the server type

```
$ bokeh-server
```

This will start a server that is accesible on your computer at `http://localhost:5006`. If you want to make sure that you can access your plots across a network (or the internet), you can listen on all IP addresses using

```
$ bokeh-server --ip 0.0.0.0
```

Now that your plotting server is up and running, start your main loop and pass the `Plot` extension. Consider this example of fitting the function $f(x) = x^a$ to $f(x) = x^2$.

```
>>> import theano
>>> a = theano.shared(3.)
>>> a.name = 'a'
>>> x = theano.tensor.scalar('data')
>>> cost = abs(x ** 2 - x ** a)
>>> cost.name = 'cost'
```

We train on a 150 random points in $[0, 1]$.

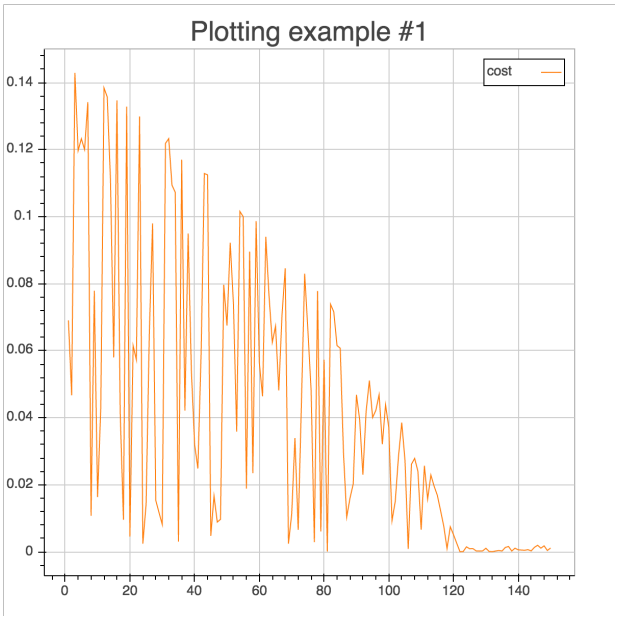
```
>>> import numpy
>>> from fuel.streams import DataStream
>>> from fuel.datasets import IterableDataset
>>> data_stream = DataStream(IterableDataset(
...     numpy.random.rand(150).astype(theano.config.floatX)))
```

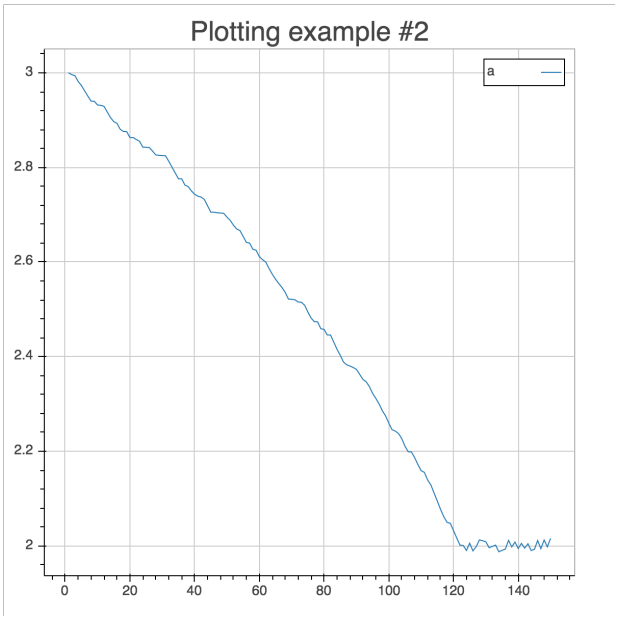
Now let's train with gradient descent and plot the results.

```
>>> from blocks.main_loop import MainLoop
>>> from blocks.algorithms import GradientDescent, Scale
>>> from blocks.extensions import FinishAfter
>>> from blocks.extensions.monitoring import TrainingDataMonitoring
>>> from blocks_extras.extensions.plot import Plot
>>> main_loop = MainLoop(
...     model=None, data_stream=data_stream,
...     algorithm=GradientDescent(cost=cost,
...                               parameters=[a],
...                               step_rule=Scale(learning_rate=0.1)),
...     extensions=[FinishAfter(after_n_epochs=1),
...                 TrainingDataMonitoring([cost, a], after_batch=True),
...                 Plot('Plotting example', channels=[['cost'], ['a']],
...                      after_batch=True)])
>>> main_loop.run()
```

Tip: If you want to plot channels in the same figure, pass them as part of the same list. For example, `[['cost', 'a']]` would have plotted a single figure with both the cost and the estimate of the exponent.

Open up your browser and go to `http://localhost:5006` to see your model cost go down in real-time!


Plotting example #1
cost
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100
120
140


Plotting example #2
a
3
2.8
2.6
2.4
2.2
2
0
20
40
60
80
100
120
140

CHAPTER 2

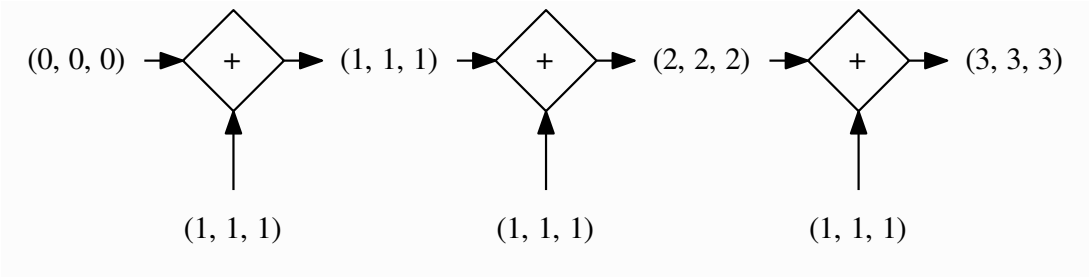
In-depth

2.1 Recurrent neural networks

Warning: This section is very much work in progress!

This tutorial explains recurrent bricks in Blocks. Readers unfamiliar with bricks should start with the *bricks overview* first and continue with this tutorial afterwards.

2.1.1 Quickstart example

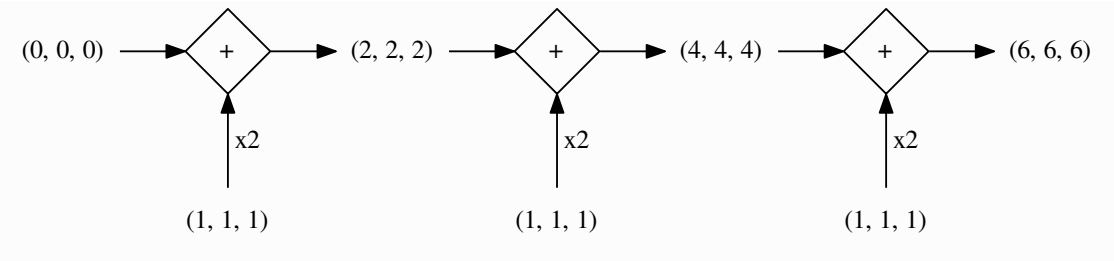


As a starting example, we'll be building an RNN which accumulates the input it receives (figure above). The equation describing that RNN is

$$\mathbf{h}_t = \mathbf{h}_{t-1} + \mathbf{x}_t$$

```
>>> import numpy
>>> import theano
>>> from theano import tensor
>>> from blocks import initialization
>>> from blocks.bricks import Identity
>>> from blocks.bricks.recurrent import SimpleRecurrent
>>> x = tensor.tensor3('x')
>>> rnn = SimpleRecurrent(
...     dim=3, activation=Identity(), weights_init=initialization.Identity())
>>> rnn.initialize()
>>> h = rnn.apply(x)
>>> f = theano.function([x], h)
>>> print(f(numpy.ones((3, 1, 3), dtype=theano.config.floatX)))
[[[ 1.  1.  1.]]

 [[ 2.  2.  2.]]

 [[ 3.  3.  3.]]]...
```

Let's modify that example so that the RNN accumulates two times the input it receives (figure below).



The equation for the RNN is

$$\mathbf{h}_t = \mathbf{h}_{t-1} + 2 \cdot \mathbf{x}_t$$

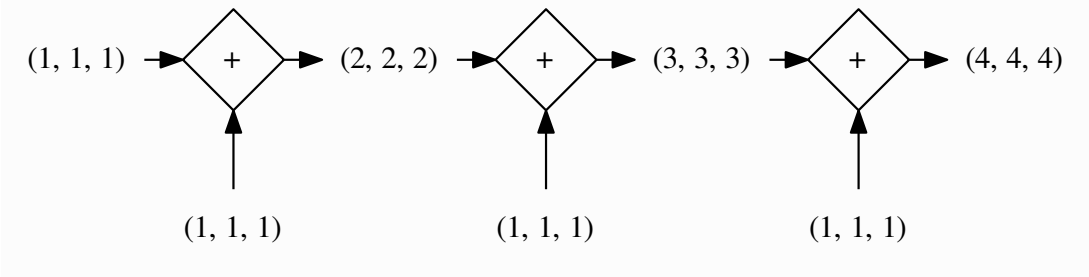
```
>>> from blocks.bricks import Linear
>>> doubler = Linear(
...     input_dim=3, output_dim=3, weights_init=initialization.Identity(2),
...     biases_init=initialization.Constant(0))
>>> doubler.initialize()
>>> h_doubler = rnn.apply(doubler.apply(x))
>>> f = theano.function([x], h_doubler)
>>> print(f(numpy.ones((3, 1, 3), dtype=theano.config.floatX)))
[[[ 2.  2.  2.]]

 [[ 4.  4.  4.]]

 [[ 6.  6.  6.]]]...
```

Note that in order to double the input we had to apply a *bricks.Linear* brick to `x`, even though

$$\mathbf{h}_t = f(\mathbf{V}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t + \mathbf{b})$$

is what is usually thought of as the RNN equation. The reason why recurrent bricks work that way is it allows greater flexibility and modularity: $\mathbf{W}\mathbf{x}_t$ can be replaced by a whole neural network if we want.

2.1.2 Initial states



Recurrent models all have in common that their initial state has to be specified. However, in constructing our toy examples, we omitted to pass $\mathbf{h}_0$ when applying the recurrent brick. What happened?

It turns out that recurrent bricks set that initial state to zero if it's not passed as argument, which is a good sane default in most cases, but we can just as well set it explicitly.

We will modify the starting example so that it accumulates the input it receives, but starting from one instead of zero (figure above):

$$\mathbf{h}_t = \mathbf{h}_{t-1} + \mathbf{x}_t, \quad \mathbf{h}_0 = 1$$

```
>>> h0 = tensor.matrix('h0')
>>> h = rnn.apply(inputs=x, states=h0)
>>> f = theano.function([x, h0], h)
>>> print(f(numpy.ones((3, 1, 3), dtype=theano.config.floatX),
...         numpy.ones((1, 3), dtype=theano.config.floatX)))
[[[ 2.  2.  2.]]

 [[ 3.  3.  3.]]

 [[ 4.  4.  4.]]]...
```

2.1.3 Reverse

Todo: Say something about the `reverse` argument

2.1.4 Getting initial states back

Todo: Say something about the `return_initial_states` argument

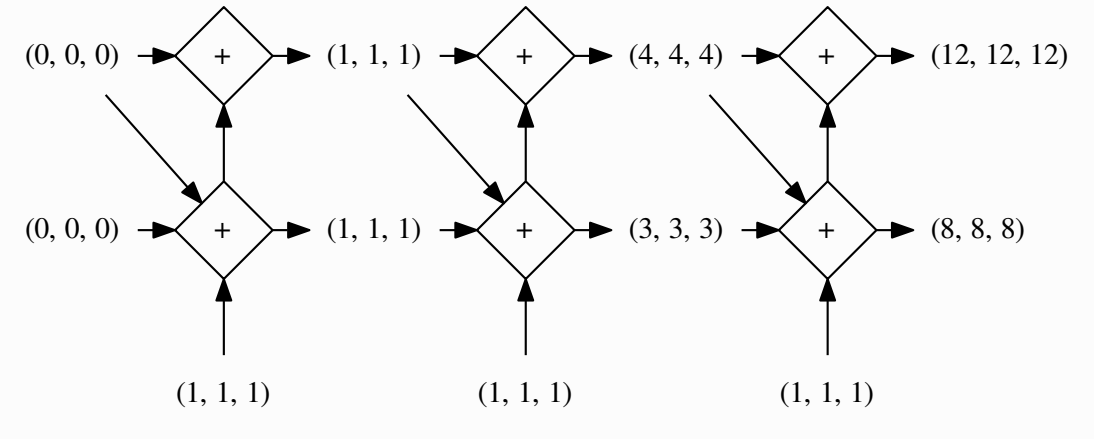
2.1.5 Iterate (or not)

The `apply` method of a recurrent brick accepts an `iterate` argument, which defaults to `True`. It is the reason for passing above a tensor of one more dimension than described in `recurrent.SimpleRecurrent.apply()` - the extra first dimension corresponds to the length of the sequence we are iterating over.

Setting `iterate` to `False` causes the `apply` method to compute only one step in the sequence.

This is very useful when you're trying to combine multiple recurrent layers in a network.

Imagine you'd like to build a network with two recurrent layers. The second layer accumulates the output of the first layer, while the first layer accumulates the input of the network and the output of the second layer (see figure below).



Here's how you can create a recurrent brick that encapsulate the two layers:

```
>>> from blocks.bricks.recurrent import BaseRecurrent, recurrent
>>> class FeedbackRNN(BaseRecurrent):
...     def __init__(self, dim, **kwargs):
...         super(FeedbackRNN, self).__init__(**kwargs)
...         self.dim = dim
...         self.first_recurrent_layer = SimpleRecurrent(
...             dim=self.dim, activation=Identity(), name='first_recurrent_layer',
...             weights_init=initialization.Identity())
...         self.second_recurrent_layer = SimpleRecurrent(
...             dim=self.dim, activation=Identity(), name='second_recurrent_layer',
...             weights_init=initialization.Identity())
...         self.children = [self.first_recurrent_layer,
...                          self.second_recurrent_layer]
...
...     @recurrent(sequences=['inputs'], contexts=[],
...                states=['first_states', 'second_states'],
...                outputs=['first_states', 'second_states'])
...     def apply(self, inputs, first_states=None, second_states=None):
...         first_h = self.first_recurrent_layer.apply(
...             inputs=inputs, states=first_states + second_states, iterate=False)
...         second_h = self.second_recurrent_layer.apply(
...             inputs=first_h, states=second_states, iterate=False)
...         return first_h, second_h
```

(continues on next page)

(continued from previous page)

```
...
...     def get_dim(self, name):
...         return (self.dim if name in ('inputs', 'first_states', 'second_states')
...                 else super(FeedbackRNN, self).get_dim(name))
...
>>> x = tensor.tensor3('x')
>>> feedback = FeedbackRNN(dim=3)
>>> feedback.initialize()
>>> first_h, second_h = feedback.apply(inputs=x)
>>> f = theano.function([x], [first_h, second_h])
>>> for states in f(numpy.ones((3, 1, 3), dtype=theano.config.floatX)):
...     print(states)
[[[ 1.  1.  1.]]

 [[ 3.  3.  3.]]

 [[ 8.  8.  8.]]]
[[[  1.   1.   1.]]

 [[  4.   4.   4.]]

 [[ 12.  12.  12.]]]...
```

There's a lot of things going on here!

We defined a recurrent brick class called `FeedbackRNN` whose constructor initializes two `bricks.recurrent.SimpleRecurrent` bricks as its children.

The class has a `get_dim` method whose purpose is to tell the dimensionality of each input to the brick's `apply` method.

The core of the class resides in its `apply` method. The `@recurrent` decorator is used to specify which of the arguments to the method are sequences to iterate over, what is returned when the method is called and which of those returned values correspond to recurrent states. Its relationship with the `inputs` and `outputs` arguments to the `@application` decorator is as follows:

- `outputs`, like in `@application`, defines everything that's returned by `apply`, including recurrent outputs
- `states` is a subset of `outputs` that corresponds to recurrent outputs, which means that the union of `sequences` and `states` forms what would be `inputs` in `@application`

Notice how no call to `theano.scan()` is being made. This is because the implementation of `apply` is responsible for computing one time step of the recurrent application of the brick. It takes states at time $t-1$ and inputs at time t and produces the output for time t. The rest is all handled by the `@recurrent` decorator behind the scenes.

This is why the `iterate` argument of the `apply` method is so useful: it allows to combine multiple recurrent brick applications within another `apply` implementation.

Tip: When looking at a recurrent brick's documentation, keep in mind that the parameters to its `apply` method are explained in terms of a single iteration, *i.e.* with the assumption that `iterate = False`.

2.1.6 See Also

- LSTM implementation: `bricks.recurrent.LSTM`
- GRU implementation: `bricks.recurrent.GatedRecurrent`

- Bidirectional RNNs: `bricks.recurrent.Bidirectional`
- Deep recurrent networks (stacked RNNs): `bricks.recurrent.RecurrentStack`

2.2 Configuration

Blocks allows module-wide configuration values to be set using a YAML configuration file and environment variables. Environment variables override the configuration file which in its turn overrides the defaults.

The configuration is read from `~/.blocksrc` if it exists. A custom configuration file can be used by setting the `BLOCKS_CONFIG` environment variable. A configuration file is of the form:

```
data_path: /home/user/datasets
```

If a setting is not configured and does not provide a default, a *ConfigurationError* is raised when it is accessed.

Configuration values can be accessed as attributes of `blocks.config.config`.

```
>>> from blocks.config import config
>>> print(config.default_seed)
1
```

The following configurations are supported:

default_seed
: The seed used when initializing random number generators (RNGs) such as NumPy `RandomState` objects as well as Theano's `MRG_RandomStreams` objects. Must be an integer. By default this is set to 1.

recursion_limit
: The recursion max depth limit used in *MainLoop* as well as in other situations when deep recursion is required. The most notable example of such a situation is pickling or unpickling a complex structure with lots of objects, such as a big Theano computation graph.

profile, BLOCKS_PROFILE
: A boolean value which determines whether to print profiling information at the end of a call to *MainLoop.run()*.

log_backend
: The backend to use for logging experiments. Defaults to *python*, which stores the log as a Python object in memory. The other option is *sqlite*.

sqlite_database, BLOCKS_SQLITEDB
: The SQLite database file to use.

max_blob_size
: The maximum size of an object to store in an SQLite database in bytes. Objects beyond this size will trigger a warning. Defaults to 4 kilobyte.

temp_dir, BLOCKS_TEMPDIR
: The directory in which Blocks will create temporary files. If unspecified, the platform-dependent default chosen by the Python `tempfile` module is used.

class `blocks.config.`**ConfigurationError**
: Bases: `exceptions.Exception`

  Error raised when a configuration value is requested but not set.

2.3 Create your own brick

This tutorial explains how to create a custom brick, which is useful if you want to group several specific operations (which can be bricks themselves) into a single one so that you can easily reuse it.

The first part of this tutorial lists the requirements and optional components that a brick should/can implement while the second part describes the construction of a simple toy brick.

This tutorial assumes that you are already familiar with *bricks* and how to use them from a user point of view.

2.3.1 Bricks ingredients and recipe

All the bricks in Blocks inherit directly or indirectly from the *Brick*. There is already a rich inheritance hierarchy of bricks implemented in Blocks and thus, you should consider which brick level to inherit from. Bear in mind that multiple inheritance is often possible and advocated whenever it makes sense.

Here are examples of possible bricks to inherit from:

- *Sequence*: a sequence of bricks.
- *Initializable*: a brick that defines a same initialization scheme (weights and biases) for all its children.
- *Feedforward*: declares an interface for bricks with one input and one output.
- *Linear*: a linear transformation with optional bias. Inherits from *Initializable* and *Feedforward*.
- *BaseRecurrent*: the base class for recurrent bricks. Check the *tutorial about rnns* for more information.
- many more!

Let's say that you want to create a brick from scratch, simply inheriting from *Brick*, then you should consider overwriting the following methods (strictly speaking, all these methods are optional, check the docstring of *Brick* for a precise description of the life-cycle of a brick):

- `Brick.__init__()`: you should pass by argument the attributes of your brick. It is also in this method that you should create the potential "children bricks" that belongs to your brick (in that case, you have to pass the children bricks to `super().__init__`). The initialization of the attributes can be lazy as described later in the tutorial.
- `apply()`: you need to implement a method that actually implements the operation of the brick, taking as arguments the inputs of the brick and returning its outputs. It can have any name and for simple bricks is often named `apply`. You should decorate it with the *application()* decorator, as explained in the next section. If you design a recurrent brick, you should instead decorate it with the *recurrent()* decorator as explained in the *tutorial about rnns*.
- *Brick._allocate()*: you should implement this method to allocate the shared variables (often representing parameters) of the brick. In Blocks, by convention, the built-in bricks allocate their shared variables with nan values and we recommend you to do the same.
- *Brick._initialize()*: you should implement this method to initialize the shared variables of your brick. This method is called after the allocation.
- *Brick._push_allocation_config()*: you should consider overwriting this method if you want to change configuration of the children bricks before they allocate their parameters.
- *Brick._push_initialization_config()*: you should consider overwriting this method if you want to change the initialization schemes of the children before they get initialized. If the children bricks need to be initialized with the same scheme, then you should inherit your brick from *Initializable*, which automatically pushes the initialization schemes of your brick (provided as arguments `weights_init` and `biases_init` of the constructor) to the children bricks.

- *get_dim()*: implementing this function is useful if you want to provide a simple way to get the dimensions of the inputs and outputs of the brick.

If you want to inherit from a specific brick, check its docstring to identify the particular methods to overwrite and the attributes to define.

Application methods

The `apply()` method listed above is probably the most important method of your brick because it is the one that actually takes theano tensors as inputs, process them and return output tensors. You should decorate it with the *application()* decorator, which names variables and register auxiliary variables of the operation you implement. It is used as follows:

```
>>> class Foo(Brick):
...     @application(inputs=['input1', 'input2'], outputs=['output'])
...     def apply(self, input1, input2):
...         y = input1 + input2
...         return y
```

In the case above, it will automatically rename the theano tensor variable `input1` to `Foo_apply_input1`, `input2` to `Foo_apply_input2` and the output of the method to `foo_apply_output`. It will also add roles and names to the tag attributes of the variables, as shown below:

```
>>> foo = Foo()
>>> i1 = tensor.matrix('i1')
>>> i2 = tensor.matrix('i2')
>>> y = foo.apply(i1, i2)
>>> theano.printing.debugprint(y)
Elemwise{identity} [id A] 'foo_apply_output'
 |Elemwise{add,no_inplace} [id B] ''
   |Elemwise{identity} [id C] 'foo_apply_input1'
   | |i1 [id D]
   |Elemwise{identity} [id E] 'foo_apply_input2'
     |i2 [id F]
>>> print(y.name)
foo_apply_output
>>> print(y.tag.name)
output
>>> print(y.tag.roles)
[OUTPUT]
```

Under the hood, the `@application` decorator creates an object of class *Application*, named `apply`, which becomes an attribute of the brick class (by opposition to class instances):

```
>>> print(type(Foo.apply))
<class 'blocks.bricks.base.Application'>
```

Application properties

In the previous examples, the names of the arguments of the application methods were directly provided as arguments of the `@application` decorator because they were common to all instances of the classes. On the other hand, if these names need to be defined differently for particular instances of the class, you should use the `apply.property` decorator. Let's say that we want to name our attribute inputs with the string `self.fancy_name`, then we should write:

```
>>> class Foo(Brick):
...     def __init__(self, fancy_name):
...         self.fancy_name = fancy_name
...     @application
...     def apply(self, input)
...         ...
...     @apply.property('inputs')
...     def apply_inputs(self):
...         # Note that you can use any python code to define the name
...         return self.fancy_name
```

Using application calls

You may want to save particular variables defined in the `apply` method in order to use them later, for example to monitor them during training. For that, you need to pass `application_call` as argument of your `apply` function and use the `add_auxiliary_variable` function to register your variables of interest, as shown in this example:

```
>>> class Foo(Brick):
...     @application
...     def apply(self, x, application_call):
...         application_call.add_auxiliary_variable(x.mean())
...         return x + 1
```

`add_auxiliary_variable` annotates the variable `x.mean()` as an auxiliary variable and you can thus later retrieve it with the computational graph *ComputationGraph* and filters *VariableFilter*. In the case of the `Foo` Brick defined above, we retrieve `x.mean()` as follows:

```
>>> from blocks.graph import ComputationGraph
>>> x = tensor.fmatrix('x')
>>> y = Foo().apply(x)
>>> cg = ComputationGraph(y)
>>> print(cg.auxiliary_variables)
[mean]
```

Lazy initialization

Instead of forcing the user to provide all the brick attributes as arguments to the `Brick.__init__()` method, you could let him/her specify them later, after the creation of the brick. To enable this mechanism, called lazy initialization, you need to decorate the constructor with the *lazy()* decorator:

```
>>> @lazy(allocation=['attr1', 'attr2'])
... def __init__(self, attr1, attr1)
...     ...
```

This allows the user to specify `attr1` and `attr2` after the creation of the brick. For example, the following `ChainOfTwoFeedforward` brick is composed of two *Feedforward* bricks for which you do not need to specify the `input_dim` of `brick2` directly at its creation.

```
>>> class ChainOfTwoFeedforward(Feedforward):
...     """Two sequential Feedforward bricks."""
...     def __init__(self, brick1, brick2, **kwargs):
...         self.brick1 = brick1
...         self.brick2 = brick2
```

(continues on next page)

(continued from previous page)

```
...         children = [self.brick1, self.brick2]
...         kwargs.setdefault('children', []).extend(children)
...         super(Feedforward, self).__init__(**kwargs)
...
...     @property
...     def input_dim(self):
...         return self.brick1.input_dim
...
...     @input_dim.setter
...     def input_dim(self, value):
...         self.brick1.input_dim = value
...
...     @property
...     def output_dim(self):
...         return self.brick2.output_dim
...
...     @output_dim.setter
...     def output_dim(self, value):
...         self.brick2.output_dim = value
...
...     def _push_allocation_config(self):
...         self.brick2.input_dim = self.brick1.get_dim('output')
...
...     @application
...     def apply(self, x):
...         return self.brick2.apply(self.brick1.apply(x))
```

Note how `get_dim` is used to retrieve the `input_dim` of `brick1`. You can now use a `ChainOfTwoFeedforward` brick as follows.

```
>>> brick1 = Linear(input_dim=3, output_dim=2, use_bias=False,
...                 weights_init=Constant(2))
>>> brick2 = Linear(output_dim=4, use_bias=False, weights_init=Constant(2))
>>>
>>> seq = ChainOfTwoFeedforward(brick1, brick2)
>>> seq.initialize()
>>> brick2.input_dim
2
```

2.3.2 Example

For the sake of the tutorial, let's consider a toy operation that takes two batch inputs and multiplies them respectively by two matrices, resulting in two outputs.

The first step is to identify which brick to inherit from. Clearly we are implementing a variant of the *Linear* brick. Contrary to *Linear*, ours has two inputs and two outputs, which means that we can not inherit from *Feedforward*, which requires a single input and a single output. Our brick will have to manage two shared variables representing the matrices to multiply the inputs with. As we want to initialize them with the same scheme, we should inherit from *Initializable*, which automatically push the initialization schemes to the children. The initialization schemes are provided as arguments `weights_init` and `biases_init` of the constructor of our brick (in the `kwargs`).

```
>>> class ParallelLinear(Initializable):
...     r"""Two linear transformations without biases.
...
...     Brick which applies two linear (affine) transformations by
```

(continues on next page)

(continued from previous page)

```
...     multiplying its two inputs with two weight matrices, resulting in
...     two outputs.
...     The two inputs, weights and outputs can have different dimensions.
...
...     Parameters
...     ----------
...     input_dim{1,2} : int
...         The dimensions of the two inputs.
...     output_dim{1,2} : int
...         The dimension of the two outputs.
...     """
...     @lazy(allocation=['input_dim1', 'input_dim2',
...                       'output_dim1', 'output_dim2'])
...     def __init__(self, input_dim1, input_dim2, output_dim1, output_dim2,
...                  **kwargs):
...         super(ParallelLinear, self).__init__(**kwargs)
...         self.input_dim1 = input_dim1
...         self.input_dim2 = input_dim2
...         self.output_dim1 = output_dim1
...         self.output_dim2 = output_dim2
...
...     def __allocate(self, input_dim, output_dim, number):
...         W = shared_floatx_nans((input_dim, output_dim),
...                                name='W'+number)
...         add_role(W, WEIGHT)
...         self.parameters.append(W)
...         self.add_auxiliary_variable(W.norm(2), name='W'+number+'_norm')
...
...     def _allocate(self):
...         self.__allocate(self.input_dim1, self.output_dim1, '1')
...         self.__allocate(self.input_dim2, self.output_dim2, '2')
...
...     def _initialize(self):
...         W1, W2 = self.parameters
...         self.weights_init.initialize(W1, self.rng)
...         self.weights_init.initialize(W2, self.rng)
...
...     @application(inputs=['input1_', 'input2_'], outputs=['output1',
...         'output2'])
...     def apply(self, input1_, input2_):
...         """Apply the two linear transformations.
...
...         Parameters
...         ----------
...         input{1,2}_ : :class:`~tensor.TensorVariable`
...             The two inputs on which to apply the transformations
...
...         Returns
...         -------
...         output{1,2} : :class:`~tensor.TensorVariable`
...             The two inputs multiplied by their respective matrices
...
...         """
...         W1, W2 = self.parameters
...         output1 = tensor.dot(input1_, W1)
...         output2 = tensor.dot(input2_, W2)
...         return output1, output2
```

(continues on next page)

(continued from previous page)

```
...
...     def get_dim(self, name):
...         if name == 'input1_':
...             return self.input_dim1
...         if name == 'input2_':
...             return self.input_dim2
...         if name == 'output1':
...             return self.output_dim1
...         if name == 'output2':
...             return self.output_dim2
...         super(ParallelLinear, self).get_dim(name)
```

You can test the brick as follows:

```
>>> input_dim1, input_dim2, output_dim1, output_dim2 = 10, 5, 2, 1
>>> batch_size1, batch_size2 = 1, 2
>>>
>>> x1_mat = 3 * numpy.ones((batch_size1, input_dim1),
...                         dtype=theano.config.floatX)
>>> x2_mat = 4 * numpy.ones((batch_size2, input_dim2),
...                         dtype=theano.config.floatX)
>>>
>>> x1 = theano.tensor.matrix('x1')
>>> x2 = theano.tensor.matrix('x2')
>>> parallel1 = ParallelLinear(input_dim1, input_dim2, output_dim1,
...                            output_dim2, weights_init=Constant(2))
>>> parallel1.initialize()
>>> output1, output2 = parallel1.apply(x1, x2)
>>>
>>> f1 = theano.function([x1, x2], [output1, output2])
>>> f1(x1_mat, x2_mat)
[array([[ 60.,  60.]]...), array([[ 40.],
       [ 40.]]...)]
```

One can also create the brick using `Linear` children bricks, which

```
>>> class ParallelLinear2(Initializable):
...     def __init__(self, input_dim1, input_dim2, output_dim1, output_dim2,
...                  **kwargs):
...         self.linear1 = Linear(input_dim1, output_dim1,
...                               use_bias=False, **kwargs)
...         self.linear2 = Linear(input_dim2, output_dim2,
...                               use_bias=False, **kwargs)
...         children = [self.linear1, self.linear2]
...         kwargs.setdefault('children', []).extend(children)
...         super(ParallelLinear2, self).__init__(**kwargs)
...
...     @application(inputs=['input1_', 'input2_'], outputs=['output1',
...         'output2'])
...     def apply(self, input1_, input2_):
...         output1 = self.linear1.apply(input1_)
...         output2 = self.linear2.apply(input2_)
...         return output1, output2
...
...     def get_dim(self, name):
...         if name in ['input1_', 'output1']:
...             return self.linear1.get_dim(name)
```

(continues on next page)

(continued from previous page)

```
...         if name in ['input2_', 'output2']:
...             return self.linear2.get_dim(name)
...         super(ParallelLinear2, self).get_dim(name)
```

You can test this new version as follows:

```
>>> parallel2 = ParallelLinear2(input_dim1, input_dim2, output_dim1,
...                             output_dim2, weights_init=Constant(2))
>>> parallel2.initialize()
>>> # The weights_init initialization scheme is pushed to the children
>>> # bricks. We can verify it as follows.
>>> w = parallel2.weights_init
>>> w0 = parallel2.children[0].weights_init
>>> w1 = parallel2.children[1].weights_init
>>> print(w == w0 == w1)
True
>>>
>>> output1, output2 = parallel2.apply(x1, x2)
>>>
>>> f2 = theano.function([x1, x2], [output1, output2])
>>> f2(x1_mat, x2_mat)
[array([[ 60.,  60.]]...), array([[ 40.],
       [ 40.]]...)]
```

Actually it was not even necessary to create a custom brick for this particular operation as Blocks has a brick, called :class:`Parallel`, which applies the same prototype brick to several inputs. In our case the prototype brick we want to apply to our two inputs is a :class:`Linear` brick with no bias:

```
>>> parallel3 = Parallel(
...     prototype=Linear(use_bias=False),
...     input_names=['input1_', 'input2_'],
...     input_dims=[input_dim1, input_dim2],
...     output_dims=[output_dim1, output_dim2], weights_init=Constant(2))
>>> parallel3.initialize()
>>>
>>> output1, output2 = parallel3.apply(x1, x2)
>>>
>>> f3 = theano.function([x1, x2], [output1, output2])
>>> f3(x1_mat, x2_mat)
[array([[ 60.,  60.]]...), array([[ 40.],
       [ 40.]]...)]
```

2.4 Serialization

The ability to save models and their training progress is important for two reasons:

1. Neural nets can take days or even weeks to train. If training is interrupted during this time, it is important that we can continue from where we left off.
2. We need the ability to save models in order to share them with others or save them for later use or inspection.

These two goals come with differing requirements, which is why Blocks implements a custom serialization approach that tries to meet both needs in the *`dump()`* and *`load()`* functions.

2.4.1 Pickling the training loop

Warning: Due to the complexity of serializing a Python objects as large as the main loop, (un)pickling will sometimes fail because it exceeds the default maximum recursion depth set in Python. Increasing the limit should fix the problem.

When checkpointing, Blocks pickles the entire `main loop`, effectively serializing the exact state of the model as well as the training state (iteration state, extensions, etc.). Technically there are some difficulties with this approach:

- Some Python objects cannot be pickled e.g. file handles, generators, dynamically generated classes, nested classes, etc.
- The pickling of Theano objects can be problematic.
- We do not want to serialize the training data kept in memory, since this can be prohibitively large.

Blocks addresses these problems by avoiding certain data structures such as generators and nested classes (see the *developer guidelines*) and overriding the pickling behaviour of some objects, making the pickling of the main loop possible.

However, pickling can be problematic for long-term storage of models, because

- Unpickling depends on the libraries used being unchanged. This means that if you updated Blocks, Theano, etc. to a new version where the interface has changed, loading your training progress could fail.
- The unpickling of Theano objects can be problematic, especially when transferring from GPU to CPU or vice versa.
- It is not possible on Python 2 to unpickle objects that were pickled in Python 3.

2.4.2 Parameter saving

This is why Blocks intercepts the pickling of all Theano shared variables (which includes the parameters), and stores them as separate NPY files. The resulting file is a ZIP archive that contains the pickled main loop as well as a collection of NumPy arrays. The NumPy arrays (and hence parameters) in the ZIP file can be read, across platforms, using the `numpy.load()` function, making it possible to inspect and load parameter values, even if the unpickling of the main loop fails.

2.5 API Reference

Warning: This API reference is currently nothing but a dump of docstrings, ordered alphabetically.

The API reference contains detailed descriptions of the different end-user classes, functions, methods, etc. you will need to work with Blocks.

Note: This API reference only contains *end-user* documentation. If you are looking to hack away at Blocks' internals, you will find more detailed comments in the source code.

2.5.1 Algorithms

class blocks.algorithms.**AdaDelta**(*decay_rate=0.95*, *epsilon=1e-06*)
Bases: *blocks.algorithms.StepRule*

Adapts the step size over time using only first order information.

Parameters

- **decay_rate** (*float, optional*) – Decay rate in [0, 1]. Defaults to 0.95.
- **epsilon** (*float, optional*) – Stabilizing constant for RMS. Defaults to 1e-6.

Notes

For more information, see *[ADADELTA]*.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of `compute_steps()`, it relieves from writing a loop each time.

Parameters

- **parameter** (`TensorSharedVariable`) – The parameter.
- **previous_step** (`TensorVariable`) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (`Variable`) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**AdaGrad**(*learning_rate=0.002*, *epsilon=1e-06*)
Bases: *blocks.algorithms.StepRule*

Implements the AdaGrad learning rule.

Parameters

- **learning_rate** (*float, optional*) – Step size. Default value is set to 0.0002.
- **epsilon** (*float, optional*) – Stabilizing constant for one over root of sum of squares. Defaults to 1e-6.

Notes

For more information, see *[ADAGRAD]*.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of `compute_steps()`, it relieves from writing a loop each time.

Parameters

- **parameter** (`TensorSharedVariable`) – The parameter.

- **previous_step** (TensorVariable) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (Variable) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**Adam**(*learning_rate=0.002*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, *decay_factor=1*)

Bases: *blocks.algorithms.StepRule*

Adam optimizer as described in *[King2014]*.

Parameters

- **learning_rate** (*float, optional*) – Step size. Default value is set to 0.002.
- **beta1** (*float, optional*) – Exponential decay rate for the first moment estimates. Default value is set to 0.9.
- **beta2** (*float, optional*) – Exponential decay rate for the second moment estimates. Default value is set to 0.999.
- **epsilon** (*float, optional*) – Default value is set to 1e-8.
- **decay_factor** (*float, optional*) – Default value is set to 1.

compute_step(*parameter*, *previous_step*)

Build a Theano expression for the step for a parameter.

This method is called by default implementation of compute_steps(), it relieves from writing a loop each time.

Parameters

- **parameter** (TensorSharedVariable) – The parameter.
- **previous_step** (TensorVariable) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (Variable) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**BasicMomentum**(*momentum=0.0*)

Bases: *blocks.algorithms.StepRule*

Accumulates step with exponential discount.

Parameters **momentum** (*float, optional*) – The momentum coefficient. Defaults to 0.

Notes

This step rule is intended to be used in conjunction with another step rule, _e.g._ *Scale*. For an all-batteries-included experience, look at *Momentum*.

compute_step(*parameter*, *previous_step*)

Build a Theano expression for the step for a parameter.

This method is called by default implementation of `compute_steps()`, it relieves from writing a loop each time.

Parameters

- **parameter** (`TensorSharedVariable`) – The parameter.
- **previous_step** (`TensorVariable`) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (`Variable`) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**BasicRMSProp**(*decay_rate=0.9*, *max_scaling=100000.0*)
Bases: *blocks.algorithms.StepRule*

Scales the step size by a running average of the recent step norms.

Parameters

- **decay_rate** (*float, optional*) – How fast the running average decays, value in [0, 1] (lower is faster). Defaults to 0.9.
- **max_scaling** (*float, optional*) – Maximum scaling of the step size, in case the running average is really small. Needs to be greater than 0. Defaults to 1e5.

Notes

This step rule is intended to be used in conjunction with another step rule, _e.g._ *Scale*. For an all-batteries-included experience, look at *RMSProp*.

In general, this step rule should be used _before_ other step rules, because it has normalization properties that may undo their work. For instance, it should be applied first when used in conjunction with *Scale*.

For more information, see *[Hint2014]*.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of `compute_steps()`, it relieves from writing a loop each time.

Parameters

- **parameter** (`TensorSharedVariable`) – The parameter.
- **previous_step** (`TensorVariable`) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (`Variable`) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**CompositeRule**(*components*)
Bases: *blocks.algorithms.StepRule*

Chains several step rules.

Parameters **components** (list of *StepRule*) – The learning rules to be chained. The rules will be applied in the order as given.

compute_steps(*previous_steps*)

Build a Theano expression for steps for all parameters.

Override this method if you want to process the steps with respect to all parameters as a whole, not parameter-wise.

Parameters **previous_steps** (*OrderedDict*) – An OrderedDict of (TensorSharedVariable TensorVariable) pairs. The keys are the parameters being trained, the values are the expressions for quantities related to gradients of the cost with respect to the parameters, either the gradients themselves or steps in related directions.

Returns

- **steps** (*OrderedDict*) – A dictionary of the proposed steps in the same form as *previous_steps*.
- **updates** (*list*) – A list of tuples representing updates to be performed.

class blocks.algorithms.**GradientDescent**(*cost=None*, *parameters=None*, *step_rule=None*, *gradients=None*, *known_grads=None*, *consider_constant=None*, ***kwargs*)

Bases: *blocks.algorithms.UpdatesAlgorithm*

A base class for all gradient descent algorithms.

By "gradient descent" we mean a training algorithm of the following form:

```
for batch in data:
    steps = step_rule.compute_steps(parameters,
                                    gradients_wr_parameters)
    for parameter in parameters:
        parameter -= steps[parameter]
```

Note, that the step is *subtracted, not added*! This is done in order to make step rule chaining possible.

Parameters

- **cost** (TensorVariable, optional) – The objective to be minimized. Unused if *gradients* is specified.
- **parameters** (list of TensorSharedVariable, optional) – The parameters to be tuned. If not provided, inferred from the keys of *gradients* (in which case *gradients must* be an *OrderedDict*).
- **step_rule** (instance of *StepRule*, optional) – An object encapsulating most of the algorithm's logic. Its *compute_steps* method is called to get Theano expression for steps. Note, that the step rule might have a state, e.g. to remember a weighted sum of gradients from previous steps like it is done in gradient descent with momentum. If None, an instance of *Scale* is created.
- **gradients** (*OrderedDict or list of 2-tuples, optional*) – A dictionary mapping a parameter to an expression for the cost's gradient with respect to the parameter, or equivalently, a list of (parameter, gradient) tuples. If None, the gradient are taken automatically using theano.gradient.grad().
- **known_grads** (*dict, optional*) – A passthrough to *theano.tensor.grad*'s *known_grads* argument. Useful when you know the [approximate] gradients of some sub-expressions and would like Theano to use that information to compute parameter gradients. Only makes sense when *gradients* is *None*.

- **consider_constant** (*list, optional*) – A passthrough to *theano.tensor.grad*'s *consider_constant* argument. A list of expressions through which gradients will not be backpropagated. Only makes sense when *gradients* is *None*.

gradients
OrderedDict – The gradient dictionary.

step_rule
instance of *StepRule* – The step rule.

Notes

Changing *updates* attribute or calling *add_updates* after the *initialize* method is called will have no effect.

If a cost and parameters are provided, gradients are taken immediately upon construction, and changes to these attributes after construction will have no effect.

gradients must be an *OrderedDict* if *parameters* is unspecified because ordinary dictionaries have an unpredictable iteration order due to hash randomization (which is enabled by default since versions 2.7.3 and 3.2.3 of Python). This source of variability, when combined with Theano's heuristic graph optimizations, can cause serious reproducibility issues.

class blocks.algorithms.**Momentum**(*learning_rate=1.0*, *momentum=0.0*)
Bases: *blocks.algorithms.CompositeRule*

Accumulates step with exponential discount.

Combines *BasicMomentum* and *Scale* to form the usual momentum step rule.

Parameters

- **learning_rate** (*float, optional*) – The learning rate by which the previous step scaled. Defaults to 1.
- **momentum** (*float, optional*) – The momentum coefficient. Defaults to 0.

learning_rate
SharedVariable – A variable for learning rate.

momentum
SharedVariable – A variable for momentum.

See also:

SharedVariableModifier

class blocks.algorithms.**RMSProp**(*learning_rate=1.0*, *decay_rate=0.9*, *max_scaling=100000.0*)
Bases: *blocks.algorithms.CompositeRule*

Scales the step size by a running average of the recent step norms.

Combines *BasicRMSProp* and *Scale* to form the step rule described in *[Hint2014]*.

Parameters

- **learning_rate** (*float, optional*) – The learning rate by which the previous step scaled. Defaults to 1.
- **decay_rate** (*float, optional*) – How fast the running average decays (lower is faster). Defaults to 0.9.
- **max_scaling** (*float, optional*) – Maximum scaling of the step size, in case the running average is really small. Defaults to 1e5.

learning_rate
SharedVariable – A variable for learning rate.

decay_rate
SharedVariable – A variable for decay rate.

See also:

SharedVariableModifier

class blocks.algorithms.**RemoveNotFinite**(*scaler=1*)
Bases: *blocks.algorithms.StepRule*

A step rule that skips steps with non-finite elements.

Replaces a step (the parameter update of a single shared variable) which contains non-finite elements (such as inf or NaN) with a step rescaling the parameters.

Parameters **scaler** (*float, optional*) – The scaling applied to the parameter in case the step contains non-finite elements. Defaults to 1, which means that parameters will not be changed.

Notes

This rule should be applied last!

This trick was originally used in the GroundHog framework.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of compute_steps(), it relieves from writing a loop each time.

Parameters

- **parameter** (TensorSharedVariable) – The parameter.
- **previous_step** (TensorVariable) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (Variable) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**Restrict**(*step_rule*, *variables*)
Bases: *blocks.algorithms.StepRule*

Applies a given *StepRule* only to certain variables.

Example applications include clipping steps on only certain parameters, or scaling a certain kind of parameter's updates (e.g. adding an additional scalar multiplier to the steps taken on convolutional filters).

Parameters

- **step_rule** (*StepRule*) – The *StepRule* to be applied on the given variables.
- **variables** (*iterable*) – A collection of Theano variables on which to apply *step_rule*. Variables not appearing in this collection will not have *step_rule* applied to them.

compute_steps(*previous_steps*)
Build a Theano expression for steps for all parameters.

Override this method if you want to process the steps with respect to all parameters as a whole, not parameter-wise.

Parameters **previous_steps** (*OrderedDict*) – An OrderedDict of (TensorSharedVariable TensorVariable) pairs. The keys are the parameters being trained, the values are the expressions for quantities related to gradients of the cost with respect to the parameters, either the gradients themselves or steps in related directions.

Returns

- **steps** (*OrderedDict*) – A dictionary of the proposed steps in the same form as *previous_steps*.
- **updates** (*list*) – A list of tuples representing updates to be performed.

class blocks.algorithms.**Scale**(*learning_rate=1.0*)
Bases: *blocks.algorithms.StepRule*

A step in the direction proportional to the previous step.

If used in *GradientDescent* alone, this step rule implements steepest descent.

Parameters **learning_rate** (*float*) – The learning rate by which the previous step is multiplied to produce the step.

learning_rate
TensorSharedVariable – The shared variable storing the learning rate used.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of compute_steps(), it relieves from writing a loop each time.

Parameters

- **parameter** (TensorSharedVariable) – The parameter.
- **previous_step** (TensorVariable) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (Variable) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

class blocks.algorithms.**StepClipping**(*threshold=None*)
Bases: *blocks.algorithms.StepRule*

Rescales an entire step if its L2 norm exceeds a threshold.

When the previous steps are the gradients, this step rule performs gradient clipping.

Parameters **threshold** (*float, optional*) – The maximum permitted L2 norm for the step. The step will be rescaled to be not higher than this quanity. If None, no rescaling will be applied.

threshold
tensor.TensorSharedVariable – The shared variable storing the clipping threshold used.

compute_steps(*previous_steps*)
Build a Theano expression for steps for all parameters.

Override this method if you want to process the steps with respect to all parameters as a whole, not parameter-wise.

Parameters **previous_steps** (*OrderedDict*) – An OrderedDict of (TensorSharedVariable TensorVariable) pairs. The keys are the parameters being trained, the values are the expressions for quantities related to gradients of the cost with respect to the parameters, either the gradients themselves or steps in related directions.

Returns

- **steps** (*OrderedDict*) – A dictionary of the proposed steps in the same form as *previous_steps*.
- **updates** (*list*) – A list of tuples representing updates to be performed.

class blocks.algorithms.**StepRule**
Bases: object

A rule to compute steps for a gradient descent algorithm.

compute_step(*parameter*, *previous_step*)
Build a Theano expression for the step for a parameter.

This method is called by default implementation of *compute_steps()*, it relieves from writing a loop each time.

Parameters

- **parameter** (TensorSharedVariable) – The parameter.
- **previous_step** (TensorVariable) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (Variable) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

compute_steps(*previous_steps*)
Build a Theano expression for steps for all parameters.

Override this method if you want to process the steps with respect to all parameters as a whole, not parameter-wise.

Parameters **previous_steps** (*OrderedDict*) – An OrderedDict of (TensorSharedVariable TensorVariable) pairs. The keys are the parameters being trained, the values are the expressions for quantities related to gradients of the cost with respect to the parameters, either the gradients themselves or steps in related directions.

Returns

- **steps** (*OrderedDict*) – A dictionary of the proposed steps in the same form as *previous_steps*.
- **updates** (*list*) – A list of tuples representing updates to be performed.

class blocks.algorithms.**TrainingAlgorithm**
Bases: object

Base class for training algorithms.

A training algorithm object has a simple life-cycle. First it is initialized by calling its *initialize()* method. At this stage, for instance, Theano functions can be compiled. After that the *process_batch()* method is repeatedly called with a batch of training data as a parameter.

initialize(***kwargs*)
Initialize the training algorithm.

process_batch(*batch*)
Process a batch of training data.

batch
dict – A dictionary of (source name, data) pairs.

class blocks.algorithms.**UpdatesAlgorithm**(*updates=None*, *theano_func_kwargs=None*, *on_unused_sources='raise'*, ***kwargs*)
Bases: *blocks.algorithms.TrainingAlgorithm*

Base class for algorithms that use Theano functions with updates.

Parameters

- **updates** (list of tuples or OrderedDict) – The updates that should be performed.
- **theano_func_kwargs** (*dict, optional*) – A passthrough to *theano.function* for additional arguments. Useful for passing *profile* or *mode* arguments to the theano function that will be compiled for the algorithm.
- **on_unused_sources** (*str, one of 'raise' (default), 'ignore', 'warn'*) – Controls behavior when not all sources in a batch are used (i.e. there is no variable with a matching name in the inputs of the computational graph of the updates).

updates
list of TensorSharedVariable updates – Updates to be done for every batch. It is required that the updates are done using the old values of optimized parameters.

Notes

Changing *updates* attribute or calling *add_updates* after the *initialize* method is called will have no effect.

add_updates(*updates*)
Add updates to the training process.

The updates will be done _before_ the parameters are changed.

Parameters **updates** (list of tuples or OrderedDict) – The updates to add.

initialize()
Initialize the training algorithm.

process_batch(*batch*)
Process a batch of training data.

batch
dict – A dictionary of (source name, data) pairs.

updates

class blocks.algorithms.**VariableClipping**(*threshold*, *axis=None*)
Bases: *blocks.algorithms.StepRule*

Clip the maximum norm of individual variables along certain axes.

This *StepRule* can be used to implement L2 norm constraints on e.g. the weight vectors of individual hidden units, convolutional filters or entire weight tensors. Combine with *Restrict* (and possibly *CompositeRule*), to apply such constraints only to certain variables and/or apply different norm constraints to different variables.

Parameters

- **threshold** (*float*) – Maximum norm for a given (portion of a) tensor.
- **axis** (*int or iterable, optional*) – An integer single axis, or an iterable collection of integer axes over which to sum in order to calculate the L2 norm. If *None* (the default), the norm is computed over all elements of the tensor.

Notes

Because of the way the *StepRule* API works, this particular rule implements norm clipping of the value *after* update in the following way: it computes `parameter - previous_step`, scales it to have (possibly axes-wise) norm(s) of at most *threshold*, then subtracts *that* value from *parameter* to yield an 'equivalent step' that respects the desired norm constraints. This procedure implicitly assumes one is doing simple (stochastic) gradient descent, and so steps computed by this step rule may not make sense for use in other contexts.

Investigations into max-norm regularization date from *[Srebro2005]*. The first appearance of this technique as a regularization method for the weight vectors of individual hidden units in feed-forward neural networks may be *[Hinton2012]*.

compute_step(*parameter*, *previous_step*)

Build a Theano expression for the step for a parameter.

This method is called by default implementation of `compute_steps()`, it relieves from writing a loop each time.

Parameters

- **parameter** (`TensorSharedVariable`) – The parameter.
- **previous_step** (`TensorVariable`) – Some quantity related to the gradient of the cost with respect to the parameter, either the gradient itself or a step in a related direction.

Returns

- **step** (`Variable`) – Theano variable for the step to take.
- **updates** (*list*) – A list of tuples representing updates to be performed. This is useful for stateful rules such as *Momentum* which need to update shared variables after itetations.

2.5.2 Bricks

- *Convolutional bricks*
- *Routing bricks*
- *Recurrent bricks*
- *Attention bricks*
- *Sequence generators*
- *Cost bricks*

blocks.bricks.**application**(**args*, ***kwargs*)

Decorator for methods that apply a brick to inputs.

Parameters

- **optional** (***kwargs,*) – The application method to wrap.
- **optional** – Attributes to attach to this application.

Notes

This decorator replaces application methods with `Application` instances. It also sets the attributes given as keyword arguments to the decorator.

Note that this decorator purposely does not wrap the original method using e.g. `wraps()` or `update_wrapper()`, since that would make the class impossible to pickle (see notes at `Application`).

Examples

```
>>> class Foo(Brick):
...     @application(inputs=['x'], outputs=['y'])
...     def apply(self, x):
...         return x + 1
...     @application
...     def other_apply(self, x):
...         return x - 1
>>> foo = Foo()
>>> Foo.apply.inputs
['x']
>>> foo.apply.outputs
['y']
>>> Foo.other_apply
<blocks.bricks.base.Application object at ...>
```

class blocks.bricks.**Brick**(*name=None*, *children=None*)

Bases: `blocks.graph.annotations.Annotation`

A brick encapsulates Theano operations with parameters.

A brick goes through the following stages:

1. Construction: The call to `__init__()` constructs a *Brick* instance with a name and creates any child bricks as well.
2. Allocation of parameters:
   (a) Allocation configuration of children: The *push_allocation_config()* method configures any children of this block.
   (b) Allocation: The *allocate()* method allocates the shared Theano variables required for the parameters. Also allocates parameters for all children.
3. The following can be done in either order:
   (a) Application: By applying the brick to a set of Theano variables a part of the computational graph of the final model is constructed.
   (b) The initialization of parameters:
      i. Initialization configuration of children: The *push_initialization_config()* method configures any children of this block.

ii. Initialization: This sets the initial values of the parameters by a call to *initialize()*, which is needed to call the final compiled Theano function. Also initializes all children.

Not all stages need to be called explicitly. Step 3(a) will automatically allocate the parameters if needed. Similarly, step 3(b.2) and 2(b) will automatically perform steps 3(b.1) and 2(a) if needed. They only need to be called separately if greater control is required. The only two methods which always need to be called are an application method to construct the computational graph, and the *initialize()* method in order to initialize the parameters.

At each different stage, a brick might need a certain set of configuration settings. All of these settings can be passed to the `__init__()` constructor. However, by default many bricks support *lazy initialization*. This means that the configuration settings can be set later.

Note: Some arguments to `__init__()` are *always* required, even when lazy initialization is enabled. Other arguments must be given before calling *allocate()*, while others yet only need to be given in order to call *initialize()*. Always read the documentation of each brick carefully.

Lazy initialization can be turned off by setting `Brick.lazy = False`. In this case, there is no need to call *initialize()* manually anymore, but all the configuration must be passed to the `__init__()` method.

Parameters **name** (*str, optional*) – The name of this brick. This can be used to filter the application of certain modifications by brick names. By default, the brick receives the name of its class (lowercased).

name
str – The name of this brick.

print_shapes
bool – `False` by default. If `True` it logs the shapes of all the input and output variables, which can be useful for debugging.

parameters
list of `TensorSharedVariable` and `None` – After calling the *allocate()* method this attribute will be populated with the shared variables storing this brick's parameters. Allows for `None` so that parameters can always be accessed at the same index, even if some parameters are only defined given a particular configuration.

children
list of bricks – The children of this brick.

allocated
bool – `False` if *allocate()* has not been called yet. `True` otherwise.

initialized
bool – `False` if *allocate()* has not been called yet. `True` otherwise.

allocation_config_pushed
bool – `False` if *allocate()* or *push_allocation_config()* hasn't been called yet. `True` otherwise.

initialization_config_pushed
bool – `False` if *initialize()* or *push_initialization_config()* hasn't been called yet. `True` otherwise.

Notes

To provide support for lazy initialization, apply the *lazy()* decorator to the `__init__()` method.

Brick implementations *must* call the `__init__()` constructor of their parent using *super(BlockImplementation, self).__init__(**kwargs)* at the *beginning* of the overriding *__init__*.

The methods `_allocate()` and `_initialize()` need to be overridden if the brick needs to allocate shared variables and initialize their values in order to function.

A brick can have any number of methods which apply the brick on Theano variables. These methods should be decorated with the *application()* decorator.

If a brick has children, they must be listed in the *children* attribute. Moreover, if the brick wants to control the configuration of its children, the `_push_allocation_config()` and `_push_initialization_config()` methods need to be overridden.

Examples

Most bricks have lazy initialization enabled.

```
>>> import theano
>>> from blocks.initialization import IsotropicGaussian, Constant
>>> from blocks.bricks import Linear
>>> linear = Linear(input_dim=5, output_dim=3,
...                 weights_init=IsotropicGaussian(),
...                 biases_init=Constant(0))
>>> x = theano.tensor.vector()
>>> linear.apply(x)  # Calls linear.allocate() automatically
linear_apply_output
>>> linear.initialize()  # Initializes the weight matrix
```

allocate()
Allocate shared variables for parameters.

Based on the current configuration of this *Brick* create Theano shared variables to store the parameters. After allocation, parameters are accessible through the *parameters* attribute.

This method calls the *allocate()* method of all children first, allowing the `_allocate()` method to override the parameters of the children if needed.

Raises `ValueError` – If the configuration of this brick is insufficient to determine the number of parameters or their dimensionality to be initialized.

Notes

This method sets the *parameters* attribute to an empty list. This is in order to ensure that calls to this method completely reset the parameters.

children

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

get_dims(*names*)
Get list of dimensions for a set of input/output variables.

Parameters **names** (*list*) – The variable names.

Returns **dims** – The dimensions of the sources.

Return type list

get_hierarchical_name(*parameter*, *delimiter='/'*)
Return hierarhical name for a parameter.

Returns a path of the form `brick1/brick2/brick3.parameter1`. The delimiter is configurable.

Parameters **delimiter** (*str*) – The delimiter used to separate brick names in the path.

get_unique_path()
Returns unique path to this brick in the application graph.

initialize()
Initialize parameters.

Intialize parameters, such as weight matrices and biases.

Notes

If the brick has not allocated its parameters yet, this method will call the *allocate()* method in order to do so.

parameters

print_shapes = False
See *Brick.print_shapes*

push_allocation_config()
Push the configuration for allocation to child bricks.

Bricks can configure their children, based on their own current configuration. This will be automatically done by a call to *allocate()*, but if you want to override the configuration of child bricks manually, then you can call this function manually.

push_initialization_config()
Push the configuration for initialization to child bricks.

Bricks can configure their children, based on their own current configuration. This will be automatically done by a call to *initialize()*, but if you want to override the configuration of child bricks manually, then you can call this function manually.

blocks.bricks.**lazy**(*allocation=None*, *initialization=None*)
Makes the initialization lazy.

This decorator allows the user to define positional arguments which will not be needed until the allocation or initialization stage of the brick. If these arguments are not passed, it will automatically replace them with a custom `None` object. It is assumed that the missing arguments can be set after initialization by setting attributes with the same name.

Parameters

- **allocation** (*list*) – A list of argument names that are needed for allocation.
- **initialization** (*list*) – A list of argument names that are needed for initialization.

Examples

```
>>> class SomeBrick(Brick):
...     @lazy(allocation=['a'], initialization=['b'])
...     def __init__(self, a, b, c='c', d=None):
...         print(a, b, c, d)
>>> brick = SomeBrick('a')
a NoneInitialization c None
>>> brick = SomeBrick(d='d', b='b')
NoneAllocation b c d
```

class blocks.bricks.**BatchNormalization**(***kwargs*)

Bases: blocks.bricks.interfaces.RNGMixin, blocks.bricks.interfaces.Feedforward

Normalizes activations, parameterizes a scale and shift.

Parameters

- **input_dim** (*int or tuple*) – Shape of a single input example. It is assumed that a batch axis will be prepended to this.
- **broadcastable** (*tuple, optional*) – Tuple of the same length as *input_dim* which specifies which of the per-example axes should be averaged over to compute means and standard deviations. For example, in order to normalize over all spatial locations in a *(batch_index, channels, height, width)* image, pass *(False, True, True)*. The batch axis is always averaged out.
- **conserve_memory** (*bool, optional*) – Use an implementation that stores less intermediate state and therefore uses less memory, at the expense of 5-10% speed. Default is *True*.
- **epsilon** (*float, optional*) – The stabilizing constant for the minibatch standard deviation computation (when the brick is run in training mode). Added to the variance inside the square root, as in the batch normalization paper.
- **scale_init** (*object, optional*) – Initialization object to use for the learned scaling parameter (γ in *[BN]*). By default, uses constant initialization of 1.
- **shift_init** (*object, optional*) – Initialization object to use for the learned shift parameter (β in *[BN]*). By default, uses constant initialization of 0.
- **mean_only** (*bool, optional*) – Perform "mean-only" batch normalization as described in *[SK2016]*.
- **learn_scale** (*bool, optional*) – Whether to include a learned scale parameter (γ in *[BN]*) in this brick. Default is *True*. Has no effect if *mean_only* is *True* (i.e. a scale parameter is never learned in mean-only mode).
- **learn_shift** (*bool, optional*) – Whether to include a learned shift parameter (β in *[BN]*) in this brick. Default is *True*.

Notes

In order for trained models to behave sensibly immediately upon upon deserialization, by default, this brick runs in *inference* mode, using a population mean and population standard deviation (initialized to zeros and ones respectively) to normalize activations. It is expected that the user will adapt these during training in some fashion, independently of the training objective, e.g. by taking a moving average of minibatch-wise statistics.

In order to *train* with batch normalization, one must obtain a training graph by transforming the original inference graph. See `apply_batch_normalization()` for a routine to transform graphs, and

batch_normalization() for a context manager that may enable shorter compile times (every instance of *BatchNormalization* is itself a context manager, entry into which causes applications to be in mini-batch "training" mode, however it is usually more convenient to use batch_normalization() to enable this behaviour for all of your graph's *BatchNormalization* bricks at once).

Note that training in inference mode should be avoided, as this brick introduces scales and shift parameters (tagged with the *PARAMETER* role) that, in the absence of batch normalization, usually makes things unstable. If you must do this, filter for and remove *BATCH_NORM_SHIFT_PARAMETER* and *BATCH_NORM_SCALE_PARAMETER* from the list of parameters you are training, and this brick should behave as a (somewhat expensive) no-op.

This Brick accepts *scale_init* and *shift_init* arguments but is *not* an instance of *Initializable*, and will therefore not receive pushed initialization config from any parent brick. In almost all cases, you will probably want to stick with the defaults (unit scale and zero offset), but you can explicitly pass one or both initializers to override this.

This has the necessary properties to be inserted into a *blocks.bricks.conv.ConvolutionalSequence* as-is, in which case the *input_dim* should be omitted at construction, to be inferred from the layer below.

apply

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

image_size

normalization_axes

num_channels

num_output_channels

output_dim

class blocks.bricks.**SpatialBatchNormalization**(***kwargs*)
Bases: blocks.bricks.bn.BatchNormalization

Convenient subclass for batch normalization across spatial inputs.

Parameters **input_dim** (*int or tuple*) – The input size of a single example. Must be length at least 2. It's assumed that the first axis of this tuple is a "channels" axis, which should not be summed over, and all remaining dimensions are spatial dimensions.

Notes

See *BatchNormalization* for more details (and additional keyword arguments).

class blocks.bricks.**BatchNormalizedMLP**(***kwargs*)
Bases: blocks.bricks.sequences.MLP

Convenient subclass for building an MLP with batch normalization.

Parameters

- **conserve_memory** (*bool, optional, by keyword only*) – See *BatchNormalization*.
- **mean_only** (*bool, optional, by keyword only*) – See *BatchNormalization*.

- **learn_scale** (*bool, optional, by keyword only*) – See *BatchNormalization*.
- **learn_shift** (*bool, optional, by keyword only*) – See *BatchNormalization*.

Notes

All other parameters are the same as *MLP*. Each activation brick is wrapped in a *Sequence* containing an appropriate *BatchNormalization* brick and the activation that follows it.

By default, the contained *Linear* bricks will not contain any biases, as they could be canceled out by the biases in the *BatchNormalization* bricks being added. Pass *use_bias* with a value of *True* if you really want this for some reason.

mean_only, *learn_scale* and *learn_shift* are pushed down to all created *BatchNormalization* bricks as allocation config.

conserve_memory
Conserve memory.

class blocks.bricks.**Feedforward**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

Declares an interface for bricks with one input and one output.

Many bricks have just one input and just one output (activations, *Linear*, *MLP*). To make such bricks interchangable in most contexts they should share an interface for configuring their input and output dimensions. This brick declares such an interface.

input_dim
int – The input dimension of the brick.

output_dim
int – The output dimension of the brick.

class blocks.bricks.**Initializable**(***kwargs*)
Bases: blocks.bricks.interfaces.RNGMixin, *blocks.bricks.base.Brick*

Base class for bricks which push parameter initialization.

Many bricks will initialize children which perform a linear transformation, often with biases. This brick allows the weights and biases initialization to be configured in the parent brick and pushed down the hierarchy.

Parameters

- **weights_init** (*object*) – A *NdarrayInitialization* instance which will be used by to initialize the weight matrix. Required by *initialize()*.
- **biases_init** (object, optional) – A *NdarrayInitialization* instance that will be used to initialize the biases. Required by *initialize()* when *use_bias* is *True*. Only supported by bricks for which *has_biases* is True.
- **use_bias** (bool, optional) – Whether to use a bias. Defaults to *True*. Required by *initialize()*. Only supported by bricks for which *has_biases* is True.
- **rng** (numpy.random.RandomState) –

has_biases
bool – False if the brick does not support biases, and only has weights_init. For an example of this, see *Bidirectional*. If this is False, the brick does not support the arguments biases_init or use_bias.

has_biases = True

class blocks.bricks.**LinearLike**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable

Initializable subclass with logic for *Linear*-like classes.

Notes

Provides *W* and *b* properties that can be overridden in subclasses to implement pre-application transformations on the weights and biases. Application methods should refer to `self.W` and `self.b` rather than accessing the parameters list directly.

This assumes a layout of the parameters list with the weights coming first and biases (if `use_bias` is True) coming second.

W

b

class blocks.bricks.**Random**(*theano_seed=None*, ***kwargs*)
Bases: *blocks.bricks.base.Brick*

A mixin class for Bricks which need Theano RNGs.

Parameters **theano_seed** (*int or list, optional*) – Seed to use for a `MRG_RandomStreams` object.

seed_rng = <mtrand.RandomState object>

theano_rng
Returns Brick's Theano RNG, or a default one.

The default seed can be set through `blocks.config`.

theano_seed

class blocks.bricks.**Linear**(***kwargs*)
Bases: blocks.bricks.interfaces.LinearLike, blocks.bricks.interfaces.Feedforward

A linear transformation with optional bias.

Brick which applies a linear (affine) transformation by multiplying the input with a weight matrix. By default, a bias term is added (see *Initializable* for information on disabling this).

Parameters

- **input_dim** (*int*) – The dimension of the input. Required by *allocate()*.
- **output_dim** (*int*) – The dimension of the output. Required by *allocate()*.

Notes

See *Initializable* for initialization parameters.

A linear transformation with bias is a matrix multiplication followed by a vector summation.

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

apply
Apply the linear transformation.

Parameters **input** (TensorVariable) – The input on which to apply the transformation

Returns **output** – The transformed input plus optional bias

Return type TensorVariable

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

class blocks.bricks.**Bias**(***kwargs*)
Bases: blocks.bricks.interfaces.Feedforward, blocks.bricks.interfaces.Initializable

Add a bias (i.e. sum with a vector).

apply
Apply the linear transformation.

Parameters **input** (TensorVariable) – The input on which to apply the transformation

Returns **output** – The transformed input plus optional bias

Return type TensorVariable

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

input_dim

output_dim

class blocks.bricks.**Maxout**(***kwargs*)
Bases: *blocks.bricks.base.Brick*

Maxout pooling transformation.

A brick that does max pooling over groups of input units. If you use this code in a research project, please cite *[GWFM13]*.

Parameters **num_pieces** (*int*) – The size of the groups the maximum is taken over.

Notes

Maxout applies a set of linear transformations to a vector and selects for each output dimension the result with the highest value.

apply
Apply the maxout transformation.

Parameters **input** (TensorVariable) – The input on which to apply the transformation

Returns **output** – The transformed input

Return type TensorVariable

class blocks.bricks.**LinearMaxout**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Feedforward

Maxout pooling following a linear transformation.

This code combines the *Linear* brick with a *Maxout* brick.

Parameters

- **input_dim** (*int*) – The dimension of the input. Required by *allocate()*.
- **output_dim** (*int*) – The dimension of the output. Required by *allocate()*.
- **num_pieces** (*int*) – The number of linear functions. Required by *allocate()*.

Notes

See *Initializable* for initialization parameters.

apply

Apply the linear transformation followed by maxout.

Parameters **input** (TensorVariable) – The input on which to apply the transformations

Returns **output** – The transformed input

Return type TensorVariable

input_dim

class blocks.bricks.**Identity**(*name=None*, *children=None*)

Bases: blocks.bricks.interfaces.Activation

Elementwise application of identity function.

apply

Apply the identity function element-wise.

Parameters **input** (TensorVariable) – Theano variable to apply identity to, element-wise.

Returns **output** – The input with the activation function applied.

Return type TensorVariable

class blocks.bricks.**Tanh**(*name=None*, *children=None*)

Bases: blocks.bricks.interfaces.Activation

Elementwise application of tanh function.

apply

Apply the tanh function element-wise.

Parameters **input** (TensorVariable) – Theano variable to apply tanh to, element-wise.

Returns **output** – The input with the activation function applied.

Return type TensorVariable

class blocks.bricks.**Logistic**(*name=None*, *children=None*)

Bases: blocks.bricks.interfaces.Activation

Elementwise application of logistic function.

apply

Apply the logistic function element-wise.

Parameters **input** (TensorVariable) – Theano variable to apply logistic to, element-wise.

Returns **output** – The input with the activation function applied.

Return type `TensorVariable`

class blocks.bricks.**Softplus**(*name=None*, *children=None*)
Bases: `blocks.bricks.interfaces.Activation`

Elementwise application of softplus function.

apply
Apply the softplus function element-wise.

Parameters **input** (`TensorVariable`) – Theano variable to apply softplus to, element-wise.

Returns **output** – The input with the activation function applied.

Return type `TensorVariable`

class blocks.bricks.**Rectifier**(*name=None*, *children=None*)
Bases: `blocks.bricks.interfaces.Activation`

Elementwise application of rectifier function.

apply
Apply the rectifier function element-wise.

Parameters **input** (`TensorVariable`) – Theano variable to apply rectifier to, element-wise.

Returns **output** – The input with the activation function applied.

Return type `TensorVariable`

class blocks.bricks.**LeakyRectifier**(*leak=0.01*, ***kwargs*)
Bases: `blocks.bricks.interfaces.Activation`

Elementwise application of leakyrectifier function.

apply
Apply the leakyrectifier function element-wise.

Parameters **input** (`TensorVariable`) – Theano variable to apply leakyrectifier to, element-wise.

Returns **output** – The input with the activation function applied.

Return type `TensorVariable`

class blocks.bricks.**Softmax**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

A softmax brick.

Works with 2-dimensional inputs only. If you need more, see *NDimensionalSoftmax*.

apply
Standard softmax.

Parameters **input** (`Variable`) – A matrix, each row contains unnormalized log-probabilities of a distribution.

Returns **output_** – A matrix with probabilities in each row for each distribution from *input_*.

Return type `Variable`

categorical_cross_entropy
Computationally stable cross-entropy for pre-softmax values.

Parameters

- **y** (TensorVariable) – In the case of a matrix argument, each row represents a probabilility distribution. In the vector case, each element represents a distribution by specifying the position of 1 in a 1-hot vector.
- **x** (TensorVariable) – A matrix, each row contains unnormalized probabilities of a distribution.

Returns **cost** – A vector of cross-entropies between respective distributions from y and x.

Return type TensorVariable

log_probabilities
Normalize log-probabilities.

Converts unnormalized log-probabilities (exponents of which do not sum to one) into actual log-probabilities (exponents of which sum to one).

Parameters **input** (Variable) – A matrix, each row contains unnormalized log-probabilities of a distribution.

Returns **output** – A matrix with normalized log-probabilities in each row for each distribution from *input_*.

Return type Variable

class blocks.bricks.**NDimensionalSoftmax**(*name=None*, *children=None*)
Bases: blocks.bricks.simple.Softmax

A wrapped brick class.

This brick was automatically constructed by wrapping *Softmax* with *WithExtraDims*.

See also:

BrickWrapper For explanation of brick wrapping.

Softmax *WithExtraDims*

apply
Wraps the application method with reshapes.

Parameters **extra_ndim** (*int, optional*) – The number of extra dimensions. Default is zero.

See also:

Softmax.apply() For documentation of the wrapped application method.

apply_delegate()

categorical_cross_entropy
Wraps the application method with reshapes.

Parameters **extra_ndim** (*int, optional*) – The number of extra dimensions. Default is zero.

See also:

Softmax.categorical_cross_entropy() For documentation of the wrapped application method.

categorical_cross_entropy_delegate()

decorators = [<blocks.bricks.wrappers.WithExtraDims object>]

log_probabilities
Wraps the application method with reshapes.

Parameters **extra_ndim** (*int, optional*) – The number of extra dimensions. Default is zero.

See also:

Softmax.log_probabilities() For documentation of the wrapped application method.

log_probabilities_delegate()

class blocks.bricks.**Sequence**(*application_methods*, ***kwargs*)
Bases: *blocks.bricks.base.Brick*

A sequence of bricks.

This brick applies a sequence of bricks, assuming that their in- and outputs are compatible.

Parameters **application_methods** (*list*) – List of *BoundApplication* or *Brick* to apply. For `Brick`s, the `.apply`` method is used.

apply

apply_inputs()

apply_outputs()

class blocks.bricks.**FeedforwardSequence**(*application_methods*, ***kwargs*)
Bases: blocks.bricks.sequences.Sequence, blocks.bricks.interfaces.Feedforward

A sequence where the first and last bricks are feedforward.

Parameters **application_methods** (*list*) – List of *BoundApplication* to apply. The first and last application method should belong to a *Feedforward* brick.

input_dim

output_dim

class blocks.bricks.**MLP**(***kwargs*)
Bases: blocks.bricks.sequences.FeedforwardSequence, blocks.bricks.interfaces.Initializable

A simple multi-layer perceptron.

Parameters

- **activations** (list of *Brick*, *BoundApplication*,) – or None A list of activations to apply after each linear transformation. Give None to not apply any activation. It is assumed that the application method to use is apply. Required for __init__().
- **dims** (*list of ints*) – A list of input dimensions, as well as the output dimension of the last layer. Required for *allocate()*.
- **prototype** (*Brick*, optional) – The transformation prototype. A copy will be created for every activation. If not provided, an instance of Linear will be used.

Notes

See *Initializable* for initialization parameters.

Note that the `weights_init`, `biases_init` (as well as `use_bias` if set to a value other than the default of `None`) configurations will overwrite those of the layers each time the *MLP* is re-initialized. For more fine-grained control, push the configuration to the child layers manually before initialization.

```
>>> from blocks.bricks import Tanh
>>> from blocks.initialization import IsotropicGaussian, Constant
>>> mlp = MLP(activations=[Tanh(), None], dims=[30, 20, 10],
...           weights_init=IsotropicGaussian(),
...           biases_init=Constant(1))
>>> mlp.push_initialization_config()  # Configure children
>>> mlp.children[0].weights_init = IsotropicGaussian(0.1)
>>> mlp.initialize()
```

input_dim

output_dim

class blocks.bricks.**WithExtraDims**
Bases: *blocks.bricks.wrappers.BrickWrapper*

Wraps a brick's applications to handle inputs with extra dimensions.

A brick can be often reused even when data has more dimensions than in the default setting. An example is a situation when one wants to apply *categorical_cross_entropy()* to temporal data, that is when an additional 'time' axis is prepended to its both *x* and *y* inputs.

This wrapper adds reshapes required to use application methods of a brick with such data by merging the extra dimensions with the first non-extra one. Two key assumptions are made: that all inputs and outputs have the same number of extra dimensions and that these extra dimensions are equal throughout all inputs and outputs.

While this might be inconvinient, the wrapped brick does not try to guess the number of extra dimensions, but demands it as an argument. The considerations of simplicity and reliability motivated this design choice. Upon availability in Blocks of a mechanism to request the expected number of dimensions for an input of a brick, this can be reconsidered.

wrap(*wrapped*, *namespace*)
Wrap an application of the base brick.

This method should be overriden to write into its *namespace* argument all required changes.

Parameters

- **mcs** (*type*) – The metaclass.
- **wrapped** (*Application*) – The application to be wrapped.
- **namespace** (*dict*) – The namespace of the class being created.

class blocks.bricks.lookup.**LookupTable**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Feedforward

Encapsulates representations of a range of integers.

This brick can be used to embed integers, e.g. word indices, into a vector space.

Parameters

- **length** (*int*) – The size of the lookup table, or in other words, one plus the maximum index for which a representation is contained.
- **dim** (*int*) – The dimensionality of representations.

Notes

See *Initializable* for initialization parameters.

W

apply
Perform lookup.

Parameters **indices** (TensorVariable) – The indices of interest. The dtype must be integer.

Returns **output** – Representations for the indices of the query. Has $k + 1$ dimensions, where k is the number of dimensions of the *indices* parameter. The last dimension stands for the representation element.

Return type TensorVariable

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

has_bias = False

input_dim

output_dim

Convolutional bricks

class blocks.bricks.conv.**AveragePooling**(**kwargs*)
Bases: *blocks.bricks.conv.Pooling*

Average pooling layer.

Parameters **include_padding** (*bool, optional*) – When calculating an average, include zeros that are the result of zero padding added by the *padding* argument. A value of *True* is only accepted if *ignore_border* is also *True*. *False* by default.

Notes

For documentation on the remainder of the arguments to this class, see *MaxPooling*.

class blocks.bricks.conv.**Convolutional**(**kwargs*)
Bases: blocks.bricks.interfaces.LinearLike

Performs a 2D convolution.

Parameters

- **filter_size** (*tuple*) – The height and width of the filter (also called *kernels*).
- **num_filters** (*int*) – Number of filters per channel.

- **num_channels** (*int*) – Number of input channels in the image. For the first layer this is normally 1 for grayscale images and 3 for color (RGB) images. For subsequent layers this is equal to the number of filters output by the previous convolutional layer. The filters are pooled over the channels.
- **batch_size** (*int, optional*) – Number of examples per batch. If given, this will be passed to Theano convolution operator, possibly resulting in faster execution.
- **image_size** (*tuple, optional*) – The height and width of the input (image or feature map). If given, this will be passed to the Theano convolution operator, resulting in possibly faster execution times.
- **step** (*tuple, optional*) – The step (or stride) with which to slide the filters over the image. Defaults to (1, 1).
- **border_mode** (*{'valid', 'full'}, optional*) – The border mode to use, see `scipy.signal.convolve2d()` for details. Defaults to 'valid'.
- **tied_biases** (*bool*) – Setting this to `False` will untie the biases, yielding a separate bias for every location at which the filter is applied. If `True`, it indicates that the biases of every filter in this layer should be shared amongst all applications of that filter. Defaults to `True`.

apply

Perform the convolution.

Parameters **input** (`TensorVariable`) – A 4D tensor with the axes representing batch size, number of channels, image height, and image width.

Returns

output – A 4D tensor of filtered images (feature maps) with dimensions representing batch size, number of filters, feature map height, and feature map width.

The height and width of the feature map depend on the border mode. For 'valid' it is `image_size - filter_size + 1` while for 'full' it is `image_size + filter_size - 1`.

Return type `TensorVariable`

static conv2d_impl(*input*, *filters*, *input_shape=None*, *filter_shape=None*, *border_mode='valid'*, *subsample=(1, 1)*, *filter_flip=True*, *image_shape=None*, *filter_dilation=(1, 1)*, *num_groups=1*, *unshared=False*, ***kwargs*)

This function will build the symbolic graph for convolving a mini-batch of a stack of 2D inputs with a set of 2D filters. The implementation is modelled after Convolutional Neural Networks (CNN).

Parameters

- **input** (*symbolic 4D tensor*) – Mini-batch of feature map stacks, of shape (batch size, input channels, input rows, input columns). See the optional parameter `input_shape`.
- **filters** (*symbolic 4D or 6D tensor*) – Set of filters used in CNN layer of shape (output channels, input channels, filter rows, filter columns) for normal convolution and (output channels, output rows, output columns, input channels, filter rows, filter columns) for unshared convolution. See the optional parameter `filter_shape`.
- **input_shape** (*None, tuple/list of len 4 or 6 of int or Constant variable*) – The shape of the input parameter. Optional, possibly used to choose an optimal implementation. You can give `None` for any element of the list to specify that this element is not known at compile time.

- **filter_shape** (*None, tuple/list of len 4 or 6 of int or Constant variable*) – The shape of the filters parameter. Optional, possibly used to choose an optimal implementation. You can give `None` for any element of the list to specify that this element is not known at compile time.
- **border_mode** (*str, int or a tuple of two ints or pairs of ints*) – Either of the following:

  'valid': apply filter wherever it completely overlaps with the input. Generates output of shape: input shape - filter shape + 1

  'full': apply filter wherever it partly overlaps with the input. Generates output of shape: input shape + filter shape - 1

  'half': pad input with a symmetric border of `filter rows // 2` rows and `filter columns // 2` columns, then perform a valid convolution. For filters with an odd number of rows and columns, this leads to the output shape being equal to the input shape.

  int: pad input with a symmetric border of zeros of the given width, then perform a valid convolution.

  (int1, int2): (for 2D) pad input with a symmetric border of int1, `int2`, then perform a valid convolution.

  (int1, (int2, int3)) or ((int1, int2), int3): (for 2D) pad input with one symmetric border of *int1*‘ or `int3`, and one asymmetric border of `(int2, int3)` or `(int1, int2)`.
- **subsample** (*tuple of len 2*) – Factor by which to subsample the output. Also called strides elsewhere.
- **filter_flip** (*bool*) – If `True`, will flip the filter rows and columns before sliding them over the input. This operation is normally referred to as a convolution, and this is the default. If `False`, the filters are not flipped and the operation is referred to as a cross-correlation.
- **image_shape** (*None, tuple/list of len 4 of int or Constant variable*) – Deprecated alias for input_shape.
- **filter_dilation** (*tuple of len 2*) – Factor by which to subsample (stride) the input. Also called dilation elsewhere.
- **num_groups** (*int*) – Divides the image, kernel and output tensors into num_groups separate groups. Each which carry out convolutions separately
- **unshared** (*bool*) – If true, then unshared or ‘locally connected’ convolution will be performed. A different filter will be used for each region of the input.
- **kwargs** (*Any other keyword arguments are accepted for backwards*) – compatibility, but will be ignored.

Returns Set of feature maps generated by convolutional layer. Tensor is of shape (batch size, output channels, output rows, output columns)

Return type Symbolic 4D tensor

Notes

If cuDNN is available, it will be used on the GPU. Otherwise, it is the *CorrMM* convolution that will be used “caffe style convolution”.

This is only supported in Theano 0.8 or the development version until it is released.

The parameter filter_dilation is an implementation of dilated convolution.

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

static get_output_shape(*image_shape*, *kernel_shape*, *border_mode*, *subsample*, *filter_dilation=None*)
This function compute the output shape of convolution operation.

Parameters

- **image_shape** (*tuple of int (symbolic or numeric) corresponding to the input*) – image shape. Its four (or five) element must correspond respectively to: batch size, number of input channels, height and width (and possibly depth) of the image. None where undefined.
- **kernel_shape** (*tuple of int (symbolic or numeric) corresponding to the*) – kernel shape. For a normal convolution, its four (for 2D convolution) or five (for 3D convolution) elements must correspond respectively to : number of output channels, number of input channels, height and width (and possibly depth) of the kernel. For an unshared 2D convolution, its six channels must correspond to : number of output channels, height and width of the output, number of input channels, height and width of the kernel. None where undefined.
- **border_mode** (*string, int (symbolic or numeric) or tuple of int (symbolic*) – or numeric) or pairs of ints. If it is a string, it must be 'valid', 'half' or 'full'. If it is a tuple, its two (or three) elements respectively correspond to the padding on height and width (and possibly depth) axis. For asymmetric padding, provide a pair of ints for each dimension.
- **subsample** (*tuple of int (symbolic or numeric) Its two or three elements*) – espectively correspond to the subsampling on height and width (and possibly depth) axis.
- **filter_dilation** (*tuple of int (symbolic or numeric) Its two or three*) – elements correspond respectively to the dilation on height and width axis.
- **– The shape of the convolution output does not depend on the 'unshared'** (*Note*) – or the 'num_groups' parameters.

Returns **output_shape** – four element must correspond respectively to: batch size, number of output channels, height and width of the image. None where undefined.

Return type tuple of int corresponding to the output image shape. Its

num_output_channels

class blocks.bricks.conv.**ConvolutionalSequence**(***kwargs*)
Bases: blocks.bricks.sequences.Sequence, blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Feedforward

A sequence of convolutional (or pooling) operations.

Parameters

- **layers** (*list*) – List of convolutional bricks (i.e. *Convolutional*, ConvolutionalActivation, or *Pooling* bricks), or application methods from such bricks. Activation bricks that operate elementwise can also be included.

- **num_channels** (*int*) – Number of input channels in the image. For the first layer this is normally 1 for grayscale images and 3 for color (RGB) images. For subsequent layers this is equal to the number of filters output by the previous convolutional layer.
- **batch_size** (*int, optional*) – Number of images in batch. If given, will be passed to theano's convolution operator resulting in possibly faster execution.
- **image_size** (*tuple, optional*) – Width and height of the input (image/featuremap). If given, will be passed to theano's convolution operator resulting in possibly faster execution.
- **border_mode** (*'valid', 'full' or None, optional*) – The border mode to use, see `scipy.signal.convolve2d()` for details. Unlike with *Convolutional*, this defaults to None, in which case no default value is pushed down to child bricks at allocation time. Child bricks will in this case need to rely on either a default border mode (usually valid) or one provided at construction and/or after construction (but before allocation).
- **tied_biases** (*bool, optional*) – Same meaning as in *Convolutional*. Defaults to `None`, in which case no value is pushed to child *Convolutional* bricks.

Notes

The passed convolutional operators should be 'lazy' constructed, that is, without specifying the batch_size, num_channels and image_size. The main feature of *ConvolutionalSequence* is that it will set the input dimensions of a layer to the output dimensions of the previous layer by the `push_allocation_config()` method.

The push behaviour of *tied_biases* mirrors that of *use_bias* or any initialization configuration: only an explicitly specified value is pushed down the hierarchy. *border_mode* also has this behaviour. The reason the *border_mode* parameter behaves the way it does is that pushing a single default *border_mode* makes it very difficult to have child bricks with different border modes. Normally, such things would be overridden after *push_allocation_config()*, but this is a particular hassle as the border mode affects the allocation parameters of every subsequent child brick in the sequence. Thus, only an explicitly specified border mode will be pushed down the hierarchy.

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

class blocks.bricks.conv.**ConvolutionalTranspose**(***kwargs*)
Bases: *blocks.bricks.conv.Convolutional*

Performs the transpose of a 2D convolution.

Parameters

- **num_filters** (*int*) – Number of filters at the *output* of the transposed convolution, i.e. the number of channels in the corresponding convolution.
- **num_channels** (*int*) – Number of channels at the *input* of the transposed convolution, i.e. the number of output filters in the corresponding convolution.
- **step** (*tuple, optional*) – The step (or stride) of the corresponding *convolution*. Defaults to (1, 1).
- **image_size** (*tuple, optional*) – Image size of the input to the *transposed* convolution, i.e. the output of the corresponding convolution. Required for tied biases. Defaults to `None`.

- **unused_edge** (*tuple, optional*) – Tuple of pixels added to the inferred height and width of the output image, whose values would be ignored in the corresponding forward convolution. Must be such that 0 <= `unused_edge[i]` <= `step[i]`. Note that this parameter is **ignored** if `original_image_size` is specified in the constructor or manually set as an attribute.
- **original_image_size** (*tuple, optional*) – The height and width of the image that forms the output of the transpose operation, which is the input of the original (non-transposed) convolution. By default, this is inferred from *image_size* to be the size that has each pixel of the original image touched by at least one filter application in the original convolution. Degenerate cases with dropped border pixels (in the original convolution) are possible, and can be manually specified via this argument. See notes below.

See also:

Convolutional For the documentation of other parameters.

Notes

By default, *original_image_size* is inferred from *image_size* as being the *minimum* size of image that could have produced this output. Let `hanging[i] = original_image_size[i] - image_size[i] * step[i]`. Any value of `hanging[i]` greater than `filter_size[i] - step[i]` will result in border pixels that are ignored by the original convolution. With this brick, any `original_image_size` such that `filter_size[i] - step[i] < hanging[i] < filter_size[i]` for all `i` can be validly specified. However, no value will be output by the transposed convolution itself for these extra hanging border pixels, and they will be determined entirely by the bias.

conv2d_impl(*input_*, *W*, *input_shape*, *subsample*, *border_mode*, *filter_shape*)

This function will build the symbolic graph for convolving a mini-batch of a stack of 2D inputs with a set of 2D filters. The implementation is modelled after Convolutional Neural Networks (CNN).

Parameters

- **input** (*symbolic 4D tensor*) – Mini-batch of feature map stacks, of shape (batch size, input channels, input rows, input columns). See the optional parameter `input_shape`.
- **filters** (*symbolic 4D or 6D tensor*) – Set of filters used in CNN layer of shape (output channels, input channels, filter rows, filter columns) for normal convolution and (output channels, output rows, output columns, input channels, filter rows, filter columns) for unshared convolution. See the optional parameter `filter_shape`.
- **input_shape** (*None, tuple/list of len 4 or 6 of int or Constant variable*) – The shape of the input parameter. Optional, possibly used to choose an optimal implementation. You can give `None` for any element of the list to specify that this element is not known at compile time.
- **filter_shape** (*None, tuple/list of len 4 or 6 of int or Constant variable*) – The shape of the filters parameter. Optional, possibly used to choose an optimal implementation. You can give `None` for any element of the list to specify that this element is not known at compile time.
- **border_mode** (*str, int or a tuple of two ints or pairs of ints*) – Either of the following:

  'valid': apply filter wherever it completely overlaps with the input. Generates output of shape: input shape - filter shape + 1

'full': apply filter wherever it partly overlaps with the input. Generates output of shape: input shape + filter shape - 1

'half': pad input with a symmetric border of `filter rows // 2` rows and `filter columns // 2` columns, then perform a valid convolution. For filters with an odd number of rows and columns, this leads to the output shape being equal to the input shape.

int: pad input with a symmetric border of zeros of the given width, then perform a valid convolution.

(int1, int2): (for 2D) pad input with a symmetric border of int1, `int2`, then perform a valid convolution.

(int1, (int2, int3)) or ((int1, int2), int3): (for 2D) pad input with one symmetric border of *int1*‘ or `int3`, and one asymmetric border of `(int2, int3)` or `(int1, int2)`.

- **subsample** (*tuple of len 2*) – Factor by which to subsample the output. Also called strides elsewhere.
- **filter_flip** (*bool*) – If `True`, will flip the filter rows and columns before sliding them over the input. This operation is normally referred to as a convolution, and this is the default. If `False`, the filters are not flipped and the operation is referred to as a cross-correlation.
- **image_shape** (*None, tuple/list of len 4 of int or Constant variable*) – Deprecated alias for input_shape.
- **filter_dilation** (*tuple of len 2*) – Factor by which to subsample (stride) the input. Also called dilation elsewhere.
- **num_groups** (*int*) – Divides the image, kernel and output tensors into num_groups separate groups. Each which carry out convolutions separately
- **unshared** (*bool*) – If true, then unshared or ‘locally connected’ convolution will be performed. A different filter will be used for each region of the input.
- **kwargs** (*Any other keyword arguments are accepted for backwards*) – compatibility, but will be ignored.

Returns Set of feature maps generated by convolutional layer. Tensor is of shape (batch size, output channels, output rows, output columns)

Return type Symbolic 4D tensor

Notes

If cuDNN is available, it will be used on the GPU. Otherwise, it is the *CorrMM* convolution that will be used “caffe style convolution”.

This is only supported in Theano 0.8 or the development version until it is released.

The parameter filter_dilation is an implementation of dilated convolution.

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

original_image_size

class blocks.bricks.conv.**Flattener**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

Flattens the input.

It may be used to pass multidimensional objects like images or feature maps of convolutional bricks into bricks which allow only two dimensional input (batch, features) like MLP.

apply

class blocks.bricks.conv.**MaxPooling**(***kwargs*)
Bases: *blocks.bricks.conv.Pooling*

Max pooling layer.

Parameters

- **pooling_size** (*tuple*) – The height and width of the pooling region i.e. this is the factor by which your input's last two dimensions will be downscaled.
- **step** (*tuple, optional*) – The vertical and horizontal shift (stride) between pooling regions. By default this is equal to *pooling_size*. Setting this to a lower number results in overlapping pooling regions.
- **input_dim** (*tuple, optional*) – A tuple of integers representing the shape of the input. The last two dimensions will be used to calculate the output dimension.
- **padding** (*tuple, optional*) – A tuple of integers representing the vertical and horizontal zero-padding to be applied to each of the top and bottom (vertical) and left and right (horizontal) edges. For example, an argument of (4, 3) will apply 4 pixels of padding to the top edge, 4 pixels of padding to the bottom edge, and 3 pixels each for the left and right edge. By default, no padding is performed.
- **ignore_border** (*bool, optional*) – Whether or not to do partial downsampling based on borders where the extent of the pooling region reaches beyond the edge of the image. If *True*, a (5, 5) image with (2, 2) pooling regions and (2, 2) step will be downsampled to shape (2, 2), otherwise it will be downsampled to (3, 3). *True* by default.

Notes

Warning: As of this writing, setting *ignore_border* to *False* with a step not equal to the pooling size will force Theano to perform pooling computations on CPU rather than GPU, even if you have specified a GPU as your computation device. Additionally, Theano will only use *[cuDNN]* (if available) for pooling computations with *ignure_border* set to *True*. You can ensure that the entire input is captured by at least one pool by using the *padding* argument to add zero padding prior to pooling being performed.

class blocks.bricks.conv.**Pooling**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Feedforward

Base Brick for pooling operations.

This should generally not be instantiated directly; see *MaxPooling*.

apply
Apply the pooling (subsampling) transformation.

Parameters **input** (TensorVariable) – An tensor with dimension greater or equal to 2. The last two dimensions will be downsampled. For example, with images this means that the last two dimensions should represent the height and width of your image.

Returns **output** – A tensor with the same number of dimensions as *input_*, but with the last two dimensions downsampled.

Return type TensorVariable

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

image_size

num_channels

num_output_channels

Routing bricks

class blocks.bricks.parallel.**Distribute**(***kwargs*)
Bases: *blocks.bricks.parallel.Fork*

Transform an input and add it to other inputs.

This brick is designed for the following scenario: one has a group of variables and another separate variable, and one needs to somehow distribute information from the latter across the former. We call that "to distribute a varible across other variables", and refer to the separate variable as "the source" and to the variables from the group as "the targets".

Given a prototype brick, a *Parallel* brick makes several copies of it (each with its own parameters). At the application time the copies are applied to the source and the transformation results are added to the targets (in the literate sense).

```
>>> from theano import tensor
>>> from blocks.initialization import Constant
>>> x = tensor.matrix('x')
>>> y = tensor.matrix('y')
>>> z = tensor.matrix('z')
>>> distribute = Distribute(target_names=['x', 'y'], source_name='z',
...                         target_dims=[2, 3], source_dim=3,
...                         weights_init=Constant(2))
>>> distribute.initialize()
>>> new_x, new_y = distribute.apply(x=x, y=y, z=z)
>>> new_x.eval({x: [[2, 2]], z: [[1, 1, 1]]})
array([[ 8.,  8.]]...
>>> new_y.eval({y: [[1, 1, 1]], z: [[1, 1, 1]]})
array([[ 7.,  7.,  7.]]...
```

Parameters

- **target_names** (*list*) – The names of the targets.
- **source_name** (*str*) – The name of the source.
- **target_dims** (*list*) – A list of target dimensions, corresponding to *target_names*.
- **source_dim** (*int*) – The dimension of the source input.

- **prototype** (*Feedforward*, optional) – The transformation prototype. A copy will be created for every input. By default a linear transformation is used.

target_dims
list

source_dim
int

Notes

See *Initializable* for initialization parameters.

apply
Distribute the source across the targets.

Parameters ****kwargs** (*dict*) – The source and the target variables.

Returns **output** – The new target variables.

Return type list

apply_inputs()

apply_outputs()

class blocks.bricks.parallel.**Fork**(***kwargs*)
Bases: *blocks.bricks.parallel.Parallel*

Several outputs from one input by applying similar transformations.

Given a prototype brick, a *Fork* brick makes several copies of it (each with its own parameters). At the application time the copies are applied to the input to produce different outputs.

A typical usecase for this brick is to produce inputs for gates of gated recurrent bricks, such as GatedRecurrent.

```
>>> from theano import tensor
>>> from blocks.initialization import Constant
>>> x = tensor.matrix('x')
>>> fork = Fork(output_names=['y', 'z'],
...             input_dim=2, output_dims=[3, 4],
...             weights_init=Constant(2), biases_init=Constant(1))
>>> fork.initialize()
>>> y, z = fork.apply(x)
>>> y.eval({x: [[1, 1]]})
array([[ 5.,  5.,  5.]]...
>>> z.eval({x: [[1, 1]]})
array([[ 5.,  5.,  5.,  5.]]...
```

Parameters

- **output_names** (*list of str*) – Names of the outputs to produce.
- **input_dim** (*int*) – The input dimension.
- **prototype** (*Feedforward*, optional) – The transformation prototype. A copy will be created for every input. By default an affine transformation is used.

input_dim
int – The input dimension.

output_dims
list – The output dimensions as a list of integers, corresponding to *output_names*.

See also:

Parallel, *Initializable*

apply

apply_outputs()

class blocks.bricks.parallel.**Merge**(***kwargs*)
Bases: *blocks.bricks.parallel.Parallel*

Merges several variables by applying a transformation and summing.

Parameters

- **input_names** (*list*) – The input names.
- **input_dims** (*list*) – The dictionary of input dimensions, keys are input names, values are dimensions.
- **output_dim** (*int*) – The output dimension of the merged variables.
- **prototype** (*Feedforward*, optional) – A transformation prototype. A copy will be created for every input. If `None`, a linear transformation is used.
- **child_prefix** (*str, optional*) – A prefix for children names. By default "transform" is used.

:param .. warning::: Note that if you want to have a bias you can pass a *Linear* brick as a *prototype*, but this will result in several redundant biases. It is a better idea to use `merge.children[0].use_bias = True`.

input_names
list – The input names.

input_dims
list – List of input dimensions corresponding to *input_names*.

output_dim
int – The output dimension.

Examples

```
>>> from theano import tensor
>>> from blocks.initialization import Constant
>>> a = tensor.matrix('a')
>>> b = tensor.matrix('b')
>>> merge = Merge(input_names=['a', 'b'], input_dims=[3, 4],
...               output_dim=2, weights_init=Constant(1.))
>>> merge.initialize()
>>> c = merge.apply(a=a, b=b)
>>> c.eval({a: [[1, 1, 1]], b: [[2, 2, 2, 2]]})
array([[ 11.,  11.]]...
```

apply

apply_inputs()

class blocks.bricks.parallel.**Parallel**(***kwargs*)

Bases: blocks.bricks.interfaces.Initializable

Apply similar transformations to several inputs.

Given a prototype brick, a *Parallel* brick makes several copies of it (each with its own parameters). At the application time every copy is applied to the respective input.

```
>>> from theano import tensor
>>> from blocks.initialization import Constant
>>> x, y = tensor.matrix('x'), tensor.matrix('y')
>>> parallel = Parallel(
...     prototype=Linear(use_bias=False),
...     input_names=['x', 'y'], input_dims=[2, 3], output_dims=[4, 5],
...     weights_init=Constant(2))
>>> parallel.initialize()
>>> new_x, new_y = parallel.apply(x=x, y=y)
>>> new_x.eval({x: [[1, 1]]})
array([[ 4.,  4.,  4.,  4.]]...
>>> new_y.eval({y: [[1, 1, 1]]})
array([[ 6.,  6.,  6.,  6.,  6.]]...
```

Parameters

- **input_names** (*list*) – The input names.
- **input_dims** (*list*) – List of input dimensions, given in the same order as *input_names*.
- **output_dims** (*list*) – List of output dimensions.
- **prototype** (*Feedforward*) – The transformation prototype. A copy will be created for every input.
- **child_prefix** (*str, optional*) – The prefix for children names. By default "transform" is used.

input_names
list – The input names.

input_dims
list – Input dimensions.

output_dims
list – Output dimensions.

Notes

See *Initializable* for initialization parameters.

apply

apply_inputs()

apply_outputs()

Recurrent bricks

Recurrent architectures

class blocks.bricks.recurrent.architectures.**GatedRecurrent**(***kwargs*)

Bases: *blocks.bricks.recurrent.base.BaseRecurrent*, blocks.bricks.interfaces.Initializable

Gated recurrent neural network.

Gated recurrent neural network (GRNN) as introduced in *[CvMG14]*. Every unit of a GRNN is equipped with update and reset gates that facilitate better gradient propagation.

Parameters

- **dim** (*int*) – The dimension of the hidden state.
- **activation** (*Brick* or None) – The brick to apply as activation. If None a *Tanh* brick is used.
- **gate_activation** (*Brick* or None) – The brick to apply as activation for gates. If None a *Logistic* brick is used.

Notes

See *Initializable* for initialization parameters.

apply

Apply the gated recurrent transition.

Parameters

- **states** (TensorVariable) – The 2 dimensional matrix of current states in the shape (batch_size, dim). Required for *one_step* usage.
- **inputs** (TensorVariable) – The 2 dimensional matrix of inputs in the shape (batch_size, dim)
- **gate_inputs** (TensorVariable) – The 2 dimensional matrix of inputs to the gates in the shape (batch_size, 2 * dim).
- **mask** (TensorVariable) – A 1D binary array in the shape (batch,) which is 1 if there is data available, 0 if not. Assumed to be 1-s only if not given.

Returns **output** – Next states of the network.

Return type TensorVariable

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

state_to_gates

state_to_state

class blocks.bricks.recurrent.architectures.**LSTM**(***kwargs*)

Bases: *blocks.bricks.recurrent.base.BaseRecurrent*, blocks.bricks.interfaces.Initializable

Long Short Term Memory.

Every unit of an LSTM is equipped with input, forget and output gates. This implementation is based on code by Mohammad Pezeshki that implements the architecture used in *[GSS03]* and *[Grav13]*. It aims to do as many computations in parallel as possible and expects the last dimension of the input to be four times the output dimension.

Unlike a vanilla LSTM as described in *[HS97]*, this model has peephole connections from the cells to the gates. The output gates receive information about the cells at the current time step, while the other gates only receive information about the cells at the previous time step. All 'peephole' weight matrices are diagonal.

Parameters

- **dim** (*int*) – The dimension of the hidden state.
- **activation** (*Brick*, optional) – The activation function. The default and by far the most popular is *Tanh*.
- **gate_activation** (*Brick* or None) – The brick to apply as activation for gates (input/output/forget). If None a *Logistic* brick is used.

Notes

See *Initializable* for initialization parameters.

apply

Apply the Long Short Term Memory transition.

Parameters

- **states** (TensorVariable) – The 2 dimensional matrix of current states in the shape (batch_size, features). Required for *one_step* usage.
- **cells** (TensorVariable) – The 2 dimensional matrix of current cells in the shape (batch_size, features). Required for *one_step* usage.
- **inputs** (TensorVariable) – The 2 dimensional matrix of inputs in the shape (batch_size, features * 4). The *inputs* needs to be four times the dimension of the LSTM brick to insure each four gates receive different transformations of the input. See *[Grav13]* equations 7 to 10 for more details. The *inputs* are then split in this order: Input gates, forget gates, cells and output gates.
- **mask** (TensorVariable) – A 1D binary array in the shape (batch,) which is 1 if there is data available, 0 if not. Assumed to be 1-s only if not given.
- **[Grav13] Graves, Alex, Generating sequences with recurrent** (**.**) – *neural networks*, arXiv preprint arXiv:1308.0850 (2013).

Returns

- **states** (TensorVariable) – Next states of the network.
- **cells** (TensorVariable) – Next cell activations of the network.

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

class blocks.bricks.recurrent.architectures.**SimpleRecurrent**(***kwargs*)
Bases: *blocks.bricks.recurrent.base.BaseRecurrent*, blocks.bricks.interfaces.Initializable

The traditional recurrent transition.

The most well-known recurrent transition: a matrix multiplication, optionally followed by a non-linearity.

Parameters

- **dim** (*int*) – The dimension of the hidden state
- **activation** (*Brick*) – The brick to apply as activation.

Notes

See *Initializable* for initialization parameters.

W

apply
Apply the simple transition.

Parameters

- **inputs** (TensorVariable) – The 2D inputs, in the shape (batch, features).
- **states** (TensorVariable) – The 2D states, in the shape (batch, features).
- **mask** (TensorVariable) – A 1D binary array in the shape (batch,) which is 1 if there is data available, 0 if not. Assumed to be 1-s only if not given.

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

Helper bricks for recurrent networks

class blocks.bricks.recurrent.misc.**Bidirectional**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable

Bidirectional network.

A bidirectional network is a combination of forward and backward recurrent networks which process inputs in different order.

Parameters **prototype** (instance of BaseRecurrent) – A prototype brick from which the forward and backward bricks are cloned.

Notes

See *Initializable* for initialization parameters.

apply
Applies forward and backward networks and concatenates outputs.

apply_delegate()

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

has_bias = False

class blocks.bricks.recurrent.misc.**RecurrentStack**(*transitions*, *fork_prototype=None*, *states_name='states'*, *skip_connections=False*, ***kwargs*)

Bases: *blocks.bricks.recurrent.base.BaseRecurrent*, blocks.bricks.interfaces.Initializable

Stack of recurrent networks.

Builds a stack of recurrent layers from a supplied list of BaseRecurrent objects. Each object must have a *sequences*, *contexts*, *states* and *outputs* parameters to its *apply* method, such as the ones required by the recurrent decorator from blocks.bricks.recurrent.

In Blocks in general each brick can have an apply method and this method has attributes that list the names of the arguments that can be passed to the method and the name of the outputs returned by the method. The attributes of the apply method of this class is made from concatenating the attributes of the apply methods of each of the transitions from which the stack is made. In order to avoid conflict, the names of the arguments appearing in the *states* and *outputs* attributes of the apply method of each layers are renamed. The names of the bottom layer are used as-is and a suffix of the form '#<n>' is added to the names from other layers, where '<n>' is the number of the layer starting from 1, used for first layer above bottom.

The *contexts* of all layers are merged into a single list of unique names, and no suffix is added. Different layers with the same context name will receive the same value.

The names that appear in *sequences* are treated in the same way as the names of *states* and *outputs* if *skip_connections* is "True". The only exception is the "mask" element that may appear in the *sequences* attribute of all layers, no suffix is added to it and all layers will receive the same mask value. If you set *skip_connections* to False then only the arguments of the *sequences* from the bottom layer will appear in the *sequences* attribute of the apply method of this class. When using this class, with *skip_connections* set to "True", you can supply all inputs to all layers using a single fork which is created with *output_names* set to the *apply.sequences* attribute of this class. For example, SequenceGenerator will create a such a fork.

Whether or not *skip_connections* is set, each layer above the bottom also receives an input (values to its *sequences* arguments) from a fork of the state of the layer below it. Not to be confused with the external fork discussed in the previous paragraph. It is assumed that all *states* attributes have a "states" argument name (this can be configured with *states_name* parameter.) The output argument with this name is forked and then added to all the elements appearing in the *sequences* of the next layer (except for "mask".) If *skip_connections* is False then this fork has a bias by default. This allows direct usage of this class with input supplied only to the first layer. But if you do supply inputs to all layers (by setting *skip_connections* to "True") then by default there is no bias and the external fork you use to supply the inputs should have its own separate bias.

Parameters

- **transitions** (*list*) – List of recurrent units to use in each layer. Each derived from BaseRecurrent Note: A suffix with layer number is added to transitions' names.
- **fork_prototype** (FeedForward, optional) – A prototype for the transformation applied to states_name from the states of each layer. The transformation is used when the *states_name* argument from the *outputs* of one layer is used as input to the *sequences* of the next layer. By default it *Linear* transformation is used, with bias if skip_connections is "False". If you supply your own prototype you have to enable/disable bias depending on the value of *skip_connections*.

- **states_name** (*string*) – In a stack of RNN the state of each layer is used as input to the next. The *states_name* identify the argument of the *states* and *outputs* attributes of each layer that should be used for this task. By default the argument is called "states". To be more precise, this is the name of the argument in the *outputs* attribute of the apply method of each transition (layer.) It is used, via fork, as the *sequences* (input) of the next layer. The same element should also appear in the *states* attribute of the apply method.
- **skip_connections** (*bool*) – By default False. When true, the *sequences* of all layers are add to the *sequences* of the apply of this class. When false only the *sequences* of the bottom layer appear in the *sequences* of the apply of this class. In this case the default fork used internally between layers has a bias (see fork_prototype.) An external code can inspect the *sequences* attribute of the apply method of this class to decide which arguments it need (and in what order.) With *skip_connections* you can control what is exposed to the externl code. If it is false then the external code is expected to supply inputs only to the bottom layer and if it is true then the external code is expected to supply inputs to all layers. There is just one small problem, the external inputs to the layers above the bottom layer are added to a fork of the state of the layer below it. As a result the output of two forks is added together and it will be problematic if both will have a bias. It is assumed that the external fork has a bias and therefore by default the internal fork will not have a bias if *skip_connections* is true.

Notes

See *BaseRecurrent* for more initialization parameters.

apply
Apply the stack of transitions.

Parameters

- **low_memory** (*bool*) – Use the slow, but also memory efficient, implementation of this code.
- ***args** (TensorVariable, optional) – Positional argumentes in the order in which they appear in *self.apply.sequences* followed by *self.apply.contexts*.
- ****kwargs** (TensorVariable) – Named argument defined in *self.apply.sequences*, *self.apply.states* or *self.apply.contexts*

Returns **outputs** – The outputs of all transitions as defined in *self.apply.outputs*

Return type (list of) TensorVariable

See also:

See docstring of this class for arguments appearing in the lists *self.apply.sequences*, *self.apply.states*, *self.apply.contexts*. See `recurrent()` : for all other parameters such as *iterate* and *return_initial_states* however *reverse* is currently not implemented.

do_apply(**args*, ***kwargs*)
Apply the stack of transitions.

This is the undecorated implementation of the apply method. A method with an @apply decoration should call this method with *iterate=True* to indicate that the iteration over all steps should be done internally by this method. A method with a @*recurrent* method should have *iterate=False* (or unset) to indicate that the iteration over all steps is done externally.

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

low_memory_apply

normal_inputs(*level*)

static split_suffix(*name*)

static suffix(*name*, *level*)

static suffixes(*names*, *level*)

Base definitions for recurrent bricks

class blocks.bricks.recurrent.base.**BaseRecurrent**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

Base class for brick with recurrent application method.

has_bias = False

initial_states
Return initial states for an application call.

Default implementation assumes that the recurrent application method is called *apply*. It fetches the state names from *apply.states* and a returns a zero matrix for each of them.

SimpleRecurrent, LSTM and GatedRecurrent override this method with trainable initial states initialized with zeros.

Parameters

- **batch_size** (*int*) – The batch size.
- ***args** – The positional arguments of the application call.
- ****kwargs** – The keyword arguments of the application call.

initial_states_outputs()

blocks.bricks.recurrent.base.**recurrent**(**args*, ***kwargs*)
Wraps an apply method to allow its iterative application.

This decorator allows you to implement only one step of a recurrent network and enjoy applying it to sequences for free. The idea behind is that its most general form information flow of an RNN can be described as follows: depending on the context and driven by input sequences the RNN updates its states and produces output sequences.

Given a method describing one step of an RNN and a specification which of its inputs are the elements of the input sequence, which are the states and which are the contexts, this decorator returns an application method which implements the whole RNN loop. The returned application method also has additional parameters, see documentation of the *recurrent_apply* inner function below.

Parameters

- **sequences** (*list of strs*) – Specifies which of the arguments are elements of input sequences.
- **states** (*list of strs*) – Specifies which of the arguments are the states.
- **contexts** (*list of strs*) – Specifies which of the arguments are the contexts.

- **outputs** (*list of strs*) – Names of the outputs. The outputs whose names match with those in the *state* parameter are interpreted as next step states.

Returns **recurrent_apply** – The new application method that applies the RNN to sequences.

Return type *Application*

See also:

The tutorial on RNNs

Attention bricks

This module defines the interface of attention mechanisms and a few concrete implementations. For a gentle introduction and usage examples see the tutorial TODO.

An attention mechanism decides to what part of the input to pay attention. It is typically used as a component of a recurrent network, though one can imagine it used in other conditions as well. When the input is big and has certain structure, for instance when it is sequence or an image, an attention mechanism can be applied to extract only information which is relevant for the network in its current state.

For the purpose of documentation clarity, we fix the following terminology in this file:

- *network* is the network, typically a recurrent one, which uses the attention mechanism.
- The network has *states*. Using this word in plural might seem weird, but some recurrent networks like `LSTM` do have several states.
- The big structured input, to which the attention mechanism is applied, is called the *attended*. When it has variable structure, e.g. a sequence of variable length, there might be a *mask* associated with it.
- The information extracted by the attention from the attended is called *glimpse*, more specifically *glimpses* because there might be a few pieces of this information.

Using this terminology, the attention mechanism computes glimpses given the states of the network and the attended.

An example: in the machine translation network from *[BCB]* the attended is a sequence of so-called annotations, that is states of a bidirectional network that was driven by word embeddings of the source sentence. The attention mechanism assigns weights to the annotations. The weighted sum of the annotations is further used by the translation network to predict the next word of the generated translation. The weights and the weighted sum are the glimpses. A generalized attention mechanism for this paper is represented here as *SequenceContentAttention*.

class blocks.bricks.attention.**AbstractAttention**(***kwargs*)

Bases: *blocks.bricks.base.Brick*

The common interface for attention bricks.

First, see the module-level docstring for terminology.

A generic attention mechanism functions as follows. Its inputs are the states of the network and the attended. Given these two it produces so-called *glimpses*, that is it extracts information from the attended which is necessary for the network in its current states

For computational reasons we separate the process described above into two stages:

1. The preprocessing stage, *preprocess()*, includes computation that do not involve the state. Those can be often performed in advance. The outcome of this stage is called *preprocessed_attended*.

2. The main stage, *take_glimpses()*, includes all the rest.

When an attention mechanism is applied sequentially, some glimpses from the previous step might be necessary to compute the new ones. A typical example for that is when the focus position from the previous step is required. In such cases *take_glimpses()* should specify such need in its interface (its docstring explains how to do

that). In addition *initial_glimpses()* should specify some sensible initialization for the glimpses to be carried over.

Todo: Only single attended is currently allowed.

preprocess() and *initial_glimpses()* might end up needing masks, which are currently not provided for them.

Parameters

- **state_names** (*list*) – The names of the network states.
- **state_dims** (*list*) – The state dimensions corresponding to *state_names*.
- **attended_dim** (*int*) – The dimension of the attended.

state_names
list

state_dims
list

attended_dim
int

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_glimpses(*batch_size*, *attended*)
Return sensible initial values for carried over glimpses.

Parameters

- **batch_size** (int or `Variable`) – The batch size.
- **attended** (`Variable`) – The attended.

Returns **initial_glimpses** – The initial values for the requested glimpses. These might simply consist of zeros or be somehow extracted from the attended.

Return type list of `Variable`

preprocess
Perform the preprocessing of the attended.

Stage 1 of the attention mechanism, see *AbstractAttention* docstring for an explanation of stages. The default implementation simply returns attended.

Parameters **attended** (`Variable`) – The attended.

Returns **preprocessed_attended** – The preprocessed attended.

Return type `Variable`

take_glimpses(*attended*, *preprocessed_attended=None*, *attended_mask=None*, ***kwargs*)
Extract glimpses from the attended given the current states.

Stage 2 of the attention mechanism, see *AbstractAttention* for an explanation of stages. If *preprocessed_attended* is not given, should trigger the stage 1.

This application method *must* declare its inputs and outputs. The glimpses to be carried over are identified by their presence in both inputs and outputs list. The attended *must* be the first input, the preprocessed attended *must* be the second one.

Parameters

- **attended** (Variable) – The attended.
- **preprocessed_attended** (Variable, optional) – The preprocessed attended computed by *preprocess()*. When not given, *preprocess()* should be called.
- **attended_mask** (Variable, optional) – The mask for the attended. This is required in the case of padded structured output, e.g. when a number of sequences are force to be the same length. The mask identifies position of the *attended* that actually contain information.
- ****kwargs** (*dict*) – Includes the states and the glimpses to be carried over from the previous step in the case when the attention mechanism is applied sequentially.

class blocks.bricks.attention.**AbstractAttentionRecurrent**(*name=None*, *children=None*)

Bases: *blocks.bricks.recurrent.base.BaseRecurrent*

The interface for attention-equipped recurrent transitions.

When a recurrent network is equipped with an attention mechanism its transition typically consists of two steps: (1) the glimpses are taken by the attention mechanism and (2) the next states are computed using the current states and the glimpses. It is required for certain usecases (such as sequence generator) that apart from a do-it-all recurrent application method interfaces for the first step and the second steps of the transition are provided.

apply(***kwargs*)
Compute next states taking glimpses on the way.

compute_states(***kwargs*)
Compute next states given current states and glimpses.

take_glimpses(***kwargs*)
Compute glimpses given the current states.

class blocks.bricks.attention.**AttentionRecurrent**(*transition*, *attention*, *distribute=None*, *add_contexts=True*, *attended_name=None*, *attended_mask_name=None*, ***kwargs*)

Bases: *blocks.bricks.attention.AbstractAttentionRecurrent*, blocks.bricks.interfaces.Initializable

Combines an attention mechanism and a recurrent transition.

This brick equips a recurrent transition with an attention mechanism. In order to do this two more contexts are added: one to be attended and a mask for it. It is also possible to use the contexts of the given recurrent transition for these purposes and not add any new ones, see *add_context* parameter.

At the beginning of each step attention mechanism produces glimpses; these glimpses together with the current states are used to compute the next state and finish the transition. In some cases glimpses from the previous steps are also necessary for the attention mechanism, e.g. in order to focus on an area close to the one from the previous step. This is also supported: such glimpses become states of the new transition.

To let the user control the way glimpses are used, this brick also takes a "distribute" brick as parameter that distributes the information from glimpses across the sequential inputs of the wrapped recurrent transition.

Parameters

- **transition** (*BaseRecurrent*) – The recurrent transition.
- **attention** (*Brick*) – The attention mechanism.
- **distribute** (*Brick*, optional) – Distributes the information from glimpses across the input sequences of the transition. By default a *Distribute* is used, and those inputs containing the "mask" substring in their name are not affected.
- **add_contexts** (*bool, optional*) – If `True`, new contexts for the attended and the attended mask are added to this transition, otherwise existing contexts of the wrapped transition are used. `True` by default.
- **attended_name** (*str*) – The name of the attended context. If `None`, "attended" or the first context of the recurrent transition is used depending on the value of *add_contents* flag.
- **attended_mask_name** (*str*) – The name of the mask for the attended context. If `None`, "attended_mask" or the second context of the recurrent transition is used depending on the value of *add_contents* flag.

Notes

See *Initializable* for initialization parameters.

Wrapping your recurrent brick with this class makes all the states mandatory. If you feel this is a limitation for you, try to make it better! This restriction does not apply to sequences and contexts: those keep being as optional as they were for your brick.

Those coming to Blocks from Groundhog might recognize that this is a *RecurrentLayerWithSearch*, but on steroids :)

apply
Preprocess a sequence attending the attended context at every step.

Preprocesses the attended context and runs *do_apply()*. See *do_apply()* documentation for further information.

apply_contexts()

apply_delegate()

compute_states
Compute current states when glimpses have already been computed.

Combines an application of the *distribute* that alter the sequential inputs of the wrapped transition and an application of the wrapped transition. All unknown keyword arguments go to the wrapped transition.

Parameters ****kwargs** – Should contain everything what *self.transition* needs and in addition the current glimpses.

Returns **current_states** – Current states computed by *self.transition*.

Return type list of `TensorVariable`

compute_states_outputs()

do_apply
Process a sequence attending the attended context every step.

In addition to the original sequence this method also requires its preprocessed version, the one computed by the *preprocess* method of the attention mechanism. Unknown keyword arguments are passed to the wrapped transition.

Parameters ****kwargs** – Should contain current inputs, previous step states, contexts, the preprocessed attended context, previous step glimpses.

Returns **outputs** – The current step states and glimpses.

Return type list of `TensorVariable`

do_apply_contexts()

do_apply_outputs()

do_apply_sequences()

do_apply_states()

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

initial_states_outputs()

take_glimpses
Compute glimpses with the attention mechanism.

A thin wrapper over *self.attention.take_glimpses*: takes care of choosing and renaming the necessary arguments.

Parameters ****kwargs** – Must contain the attended, previous step states and glimpses. Can optionaly contain the attended mask and the preprocessed attended.

Returns **glimpses** – Current step glimpses.

Return type list of `TensorVariable`

take_glimpses_outputs()

class blocks.bricks.attention.**GenericSequenceAttention**(***kwargs*)
Bases: *blocks.bricks.attention.AbstractAttention*

Logic common for sequence attention mechanisms.

compute_weighted_averages
Compute weighted averages of the attended sequence vectors.

Parameters

- **weights** (`Variable`) – The weights. The shape must be equal to the attended shape without the last dimension.
- **attended** (`Variable`) – The attended. The index in the sequence must be the first dimension.

Returns **weighted_averages** – The weighted averages of the attended elements. The shape is equal to the attended shape with the first dimension dropped.

Return type `Variable`

compute_weights
Compute weights from energies in softmax-like fashion.

Todo: Use *Softmax*.

Parameters

- **energies** (Variable) – The energies. Must be of the same shape as the mask.
- **attended_mask** (Variable) – The mask for the attended. The index in the sequence must be the first dimension.

Returns **weights** – Summing to 1 non-negative weights of the same shape as *energies*.

Return type Variable

class blocks.bricks.attention.**SequenceContentAttention**(***kwargs*)

Bases: *blocks.bricks.attention.GenericSequenceAttention*, blocks.bricks.interfaces.Initializable

Attention mechanism that looks for relevant content in a sequence.

This is the attention mechanism used in *[BCB]*. The idea in a nutshell:

1. The states and the sequence are transformed independently,
2. The transformed states are summed with every transformed sequence element to obtain *match vectors*,
3. A match vector is transformed into a single number interpreted as *energy*,
4. Energies are normalized in softmax-like fashion. The resulting summing to one weights are called *attention weights*,
5. Weighted average of the sequence elements with attention weights is computed.

In terms of the *AbstractAttention* documentation, the sequence is the attended. The weighted averages from 5 and the attention weights from 4 form the set of glimpses produced by this attention mechanism.

Parameters

- **state_names** (*list of str*) – The names of the network states.
- **attended_dim** (*int*) – The dimension of the sequence elements.
- **match_dim** (*int*) – The dimension of the match vector.
- **state_transformer** (*Brick*) – A prototype for state transformations. If None, a linear transformation is used.
- **attended_transformer** (*Feedforward*) – The transformation to be applied to the sequence. If None an affine transformation is used.
- **energy_computer** (*Feedforward*) – Computes energy from the match vector. If None, an affine transformations preceeded by $tanh$ is used.

Notes

See *Initializable* for initialization parameters.

compute_energies

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_glimpses

preprocess

Preprocess the sequence for computing attention weights.

Parameters **attended** (TensorVariable) – The attended sequence, time is the 1-st dimension.

take_glimpses
Compute attention weights and produce glimpses.

Parameters

- **attended** (TensorVariable) – The sequence, time is the 1-st dimension.
- **preprocessed_attended** (TensorVariable) – The preprocessed sequence. If None, is computed by calling *preprocess()*.
- **attended_mask** (TensorVariable) – A 0/1 mask specifying available data. 0 means that the corresponding sequence element is fake.
- ****states** – The states of the network.

Returns

- **weighted_averages** (Variable) – Linear combinations of sequence elements with the attention weights.
- **weights** (Variable) – The attention weights. The first dimension is batch, the second is time.

take_glimpses_inputs()

class blocks.bricks.attention.**ShallowEnergyComputer**(***kwargs*)
Bases: blocks.bricks.sequences.Sequence, blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Feedforward

A simple energy computer: first tanh, then weighted sum.

Parameters **use_bias** (*bool, optional*) – Whether a bias should be added to the energies. Does not change anything if softmax normalization is used to produce the attention weights, but might be useful when e.g. spherical softmax is used.

input_dim

output_dim

Sequence generators

Recurrent networks are often used to generate/model sequences. Examples include language modelling, machine translation, handwriting synthesis, etc.. A typical pattern in this context is that sequence elements are generated one often another, and every generated element is fed back into the recurrent network state. Sometimes also an attention mechanism is used to condition sequence generation on some structured input like another sequence or an image.

This module provides *SequenceGenerator* that builds a sequence generating network from three main components:

- a core recurrent transition, e.g. LSTM or GatedRecurrent
- a readout component that can produce sequence elements using the network state and the information from the attention mechanism
- an attention mechanism (see *attention* for more information)

Implementation-wise *SequenceGenerator* fully relies on *BaseSequenceGenerator*. At the level of the latter an attention is mandatory, moreover it must be a part of the recurrent transition (see *AttentionRecurrent*). To simulate optional attention, *SequenceGenerator* wraps the pure recurrent network in *FakeAttentionRecurrent*.

class blocks.bricks.sequence_generators.**AbstractEmitter**(*name=None*, *children=None*)

Bases: *blocks.bricks.base.Brick*

The interface for the emitter component of a readout.

readout_dim
int – The dimension of the readout. Is given by the *Readout* brick when allocation configuration is pushed.

See also:

Readout

SoftmaxEmitter for integer outputs

Notes

An important detail about the emitter cost is that it will be evaluated with inputs of different dimensions so it has to be flexible enough to handle this. The two ways in which it can be applied are:

1. In :meth:BaseSequenceGenerator.cost_matrix where it will be applied to the whole sequence at once.

2. In :meth:BaseSequenceGenerator.generate where it will be applied to only one step of the sequence.

cost(*readouts*, *outputs*)
Implements the respective method of *Readout*.

emit(*readouts*)
Implements the respective method of *Readout*.

initial_outputs(*batch_size*)
Implements the respective method of *Readout*.

class blocks.bricks.sequence_generators.**AbstractFeedback**(*name=None*, *children=None*)

Bases: *blocks.bricks.base.Brick*

The interface for the feedback component of a readout.

See also:

Readout, *LookupFeedback*

feedback(*outputs*)
Implements the respective method of *Readout*.

class blocks.bricks.sequence_generators.**AbstractReadout**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable

The interface for the readout component of a sequence generator.

The readout component of a sequence generator is a bridge between the core recurrent network and the output sequence.

Parameters

- **source_names** (*list*) – A list of the source names (outputs) that are needed for the readout part e.g. `['states']` or `['states', 'weighted_averages']` or `['states', 'feedback']`.
- **readout_dim** (*int*) – The dimension of the readout.

source_names
list

readout_dim
int

See also:

BaseSequenceGenerator see how exactly a readout is used

Readout the typically used readout brick

cost(*readouts*, *outputs*)
Compute generation cost of outputs given readouts.

Parameters

- **readouts** (Variable) – Readouts produced by the *readout()* method of a *(..., readout dim)* shape.
- **outputs** (Variable) – Outputs whose cost should be computed. Should have as many or one less dimensions compared to *readout*. If readout has *n* dimensions, first *n - 1* dimensions of *outputs* should match with those of *readouts*.

emit(*readouts*)
Produce outputs from readouts.

Parameters **readouts** (Variable) – Readouts produced by the *readout()* method of a *(batch_size, readout_dim)* shape.

feedback(*outputs*)
Feeds outputs back to be used as inputs of the transition.

initial_outputs(*batch_size*)
Compute initial outputs for the generator's first step.

In the notation from the *BaseSequenceGenerator* documentation this method should compute y_0.

readout(***kwargs*)
Compute the readout vector from states, glimpses, etc.

Parameters ****kwargs** (*dict*) – Contains sequence generator states, glimpses, contexts and feedback from the previous outputs.

class blocks.bricks.sequence_generators.**BaseSequenceGenerator**(***kwargs*)
Bases: blocks.bricks.interfaces.Initializable

A generic sequence generator.

This class combines two components, a readout network and an attention-equipped recurrent transition, into a context-dependent sequence generator. Third component must be also given which forks feedback from the readout network to obtain inputs for the transition.

The class provides two methods: *generate()* and *cost()*. The former is to actually generate sequences and the latter is to compute the cost of generating given sequences.

The generation algorithm description follows.

Definitions and notation:

- States s_i of the generator are the states of the transition as specified in *transition.state_names*.
- Contexts of the generator are the contexts of the transition as specified in *transition.context_names*.

- Glimpses g_i are intermediate entities computed at every generation step from states, contexts and the previous step glimpses. They are computed in the transition's *apply* method when not given or by explicitly calling the transition's *take_glimpses* method. The set of glimpses considered is specified in *transition.glimpse_names*.
- Outputs y_i are produced at every step and form the output sequence. A generation cost c_i is assigned to each output.

Algorithm:

1. Initialization.

$$y_0 = readout.initial\_outputs(contexts)$$
$$s_0, g_0 = transition.initial\_states(contexts)$$
$$i = 1$$

   By default all recurrent bricks from `recurrent` have trainable initial states initialized with zeros. Subclass them or `BaseRecurrent` directly to get custom initial states.

2. New glimpses are computed:

$$g_i = transition.take\_glimpses(s_{i-1}, g_{i-1}, contexts)$$

3. A new output is generated by the readout and its cost is computed:

$$f_{i-1} = readout.feedback(y_{i-1})$$
$$r_i = readout.readout(f_{i-1}, s_{i-1}, g_i, contexts)$$
$$y_i = readout.emit(r_i)$$
$$c_i = readout.cost(r_i, y_i)$$

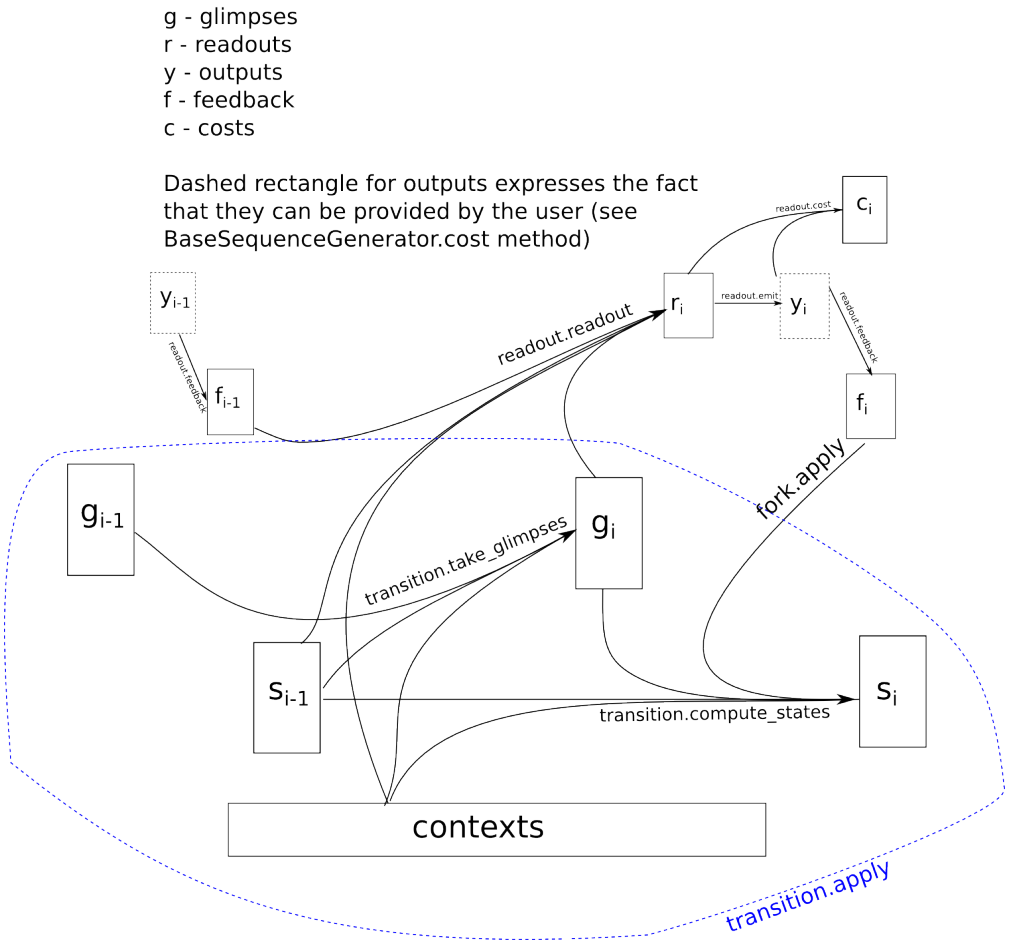
   Note that the *new* glimpses and the *old* states are used at this step. The reason for not merging all readout methods into one is to make an efficient implementation of `cost()` possible.

4. New states are computed and iteration is done:

$$f_i = readout.feedback(y_i)$$
$$s_i = transition.compute\_states(s_{i-1}, g_i, fork.apply(f_i), contexts)$$
$$i = i + 1$$

5. Back to step 2 if the desired sequence length has not been yet reached.

A scheme of the algorithm described above follows.



Parameters

- **readout** (instance of *AbstractReadout*) – The readout component of the sequence generator.
- **transition** (instance of AbstractAttentionRecurrent) – The transition component of the sequence generator.
- **fork** (*Brick*) – The brick to compute the transition's inputs from the feedback.

See also:

Initializable for initialization parameters

SequenceGenerator more user friendly interface to thisbrick

cost

Returns the average cost over the minibatch.

The cost is computed by averaging the sum of per token costs for each sequence over the minibatch.

> **Warning:** Note that, the computed cost can be problematic when batches consist of vastly different sequence lengths.

Parameters

- **outputs** (TensorVariable) – The 3(2) dimensional tensor containing output sequences. The axis 0 must stand for time, the axis 1 for the position in the batch.
- **mask** (TensorVariable) – The binary matrix identifying fake outputs.

Returns **cost** – Theano variable for cost, computed by summing over timesteps and then averaging over the minibatch.

Return type Variable

Notes

The contexts are expected as keyword arguments.

Adds average cost per sequence element *AUXILIARY* variable to the computational graph with name per_sequence_element.

cost_matrix
Returns generation costs for output sequences.

See also:

cost() Scalar cost.

generate
A sequence generation step.

Parameters **outputs** (TensorVariable) – The outputs from the previous step.

Notes

The contexts, previous states and glimpses are expected as keyword arguments.

generate_delegate()

generate_outputs()

generate_states()

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

initial_states_outputs()

class blocks.bricks.sequence_generators.**FakeAttentionRecurrent**(*transition*, ***kwargs*)

Bases: *blocks.bricks.attention.AbstractAttentionRecurrent*, blocks.bricks.interfaces.Initializable

Adds fake attention interface to a transition.

BaseSequenceGenerator requires its transition brick to support *AbstractAttentionRecurrent* interface, that is to have an embedded attention mechanism. For the cases when no attention is required (e.g.

language modeling or encoder-decoder models), *FakeAttentionRecurrent* is used to wrap a usual recurrent brick. The resulting brick has no glimpses and simply passes all states and contexts to the wrapped one.

Todo: Get rid of this brick and support attention-less transitions in *BaseSequenceGenerator*.

---

apply

apply_delegate()

compute_states

compute_states_delegate()

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_states

initial_states_outputs()

take_glimpses

class blocks.bricks.sequence_generators.**LookupFeedback**(*num_outputs=None*, *feedback_dim=None*, ***kwargs*)

Bases: *blocks.bricks.sequence_generators.AbstractFeedback*, blocks.bricks.interfaces.Initializable

A feedback brick for the case when readout are integers.

Stores and retrieves distributed representations of integers.

feedback

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

class blocks.bricks.sequence_generators.**Readout**(*emitter=None*, *feedback_brick=None*, *merge=None*, *merge_prototype=None*, *post_merge=None*, *merged_dim=None*, ***kwargs*)

Bases: *blocks.bricks.sequence_generators.AbstractReadout*

Readout brick with separated emitter and feedback parts.

Readout combines a few bits and pieces into an object that can be used as the readout component in *BaseSequenceGenerator*. This includes an emitter brick, to which *emit()*, *cost()* and *initial_outputs()* calls are delegated, a feedback brick to which *feedback()* functionality is delegated, and a pipeline to actually compute readouts from all the sources (see the *source_names* attribute of *AbstractReadout*).

The readout computation pipeline is constructed from *merge* and *post_merge* brick, whose responsibilites are described in the respective docstrings.

Parameters

- **emitter** (an instance of *AbstractEmitter*) – The emitter component.
- **feedback_brick** (an instance of *AbstractFeedback*) – The feedback component.

- **merge** (*Brick*, optional) – A brick that takes the sources given in *source_names* as an input and combines them into a single output. If given, *merge_prototype* cannot be given.
- **merge_prototype** (FeedForward, optional) – If *merge* isn't given, the transformation given by *merge_prototype* is applied to each input before being summed. By default a *Linear* transformation without biases is used. If given, *merge* cannot be given.
- **post_merge** (*Feedforward*, optional) – This transformation is applied to the merged inputs. By default *Bias* is used.
- **merged_dim** (*int, optional*) – The input dimension of *post_merge* i.e. the output dimension of *merge* (or *merge_prototype*). If not give, it is assumed to be the same as *readout_dim* (i.e. *post_merge* is assumed to not change dimensions).
- ****kwargs** (*dict*) – Passed to the parent's constructor.

See also:

BaseSequenceGenerator see how exactly a readout is used

AbstractEmitter, *AbstractFeedback*

cost

emit

feedback

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_outputs

readout

class blocks.bricks.sequence_generators.**SequenceGenerator**(*readout*, *transition*, *attention=None*, *add_contexts=True*, ***kwargs*)

Bases: *blocks.bricks.sequence_generators.BaseSequenceGenerator*

A more user-friendly interface for *BaseSequenceGenerator*.

Parameters

- **readout** (instance of *AbstractReadout*) – The readout component for the sequence generator.
- **transition** (instance of *BaseRecurrent*) – The recurrent transition to be used in the sequence generator. Will be combined with *attention*, if that one is given.
- **attention** (*object, optional*) – The attention mechanism to be added to transition, an instance of *AbstractAttention*.
- **add_contexts** (*bool*) – If True, the *AttentionRecurrent* wrapping the *transition* will add additional contexts for the attended and its mask.
- ****kwargs** (*dict*) – All keywords arguments are passed to the base class. If *fork* keyword argument is not provided, *Fork* is created that forks all transition sequential inputs without a "mask" substring in them.

class blocks.bricks.sequence_generators.**SoftmaxEmitter**(*initial_output=0*, ***kwargs*)

Bases: *blocks.bricks.sequence_generators.AbstractEmitter*, blocks.bricks.interfaces.Initializable, blocks.bricks.interfaces.Random

A softmax emitter for the case of integer outputs.

Interprets readout elements as energies corresponding to their indices.

Parameters **initial_output** (int or a scalar `Variable`) – The initial output.

cost

emit

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_outputs

probs

class blocks.bricks.sequence_generators.**TrivialEmitter**(***kwargs*)

Bases: *blocks.bricks.sequence_generators.AbstractEmitter*

An emitter for the trivial case when readouts are outputs.

Parameters **readout_dim** (*int*) – The dimension of the readout.

Notes

By default *cost()* always returns zero tensor.

cost

emit

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

initial_outputs

class blocks.bricks.sequence_generators.**TrivialFeedback**(***kwargs*)

Bases: *blocks.bricks.sequence_generators.AbstractFeedback*

A feedback brick for the case when readout are outputs.

feedback

get_dim(*name*)

Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

Cost bricks

class blocks.bricks.cost.**AbsoluteError**(*name=None*, *children=None*)

Bases: *blocks.bricks.cost.CostMatrix*

cost_matrix

class blocks.bricks.cost.**BinaryCrossEntropy**(*name=None*, *children=None*)
Bases: *blocks.bricks.cost.CostMatrix*

cost_matrix

class blocks.bricks.cost.**CategoricalCrossEntropy**(*name=None*, *children=None*)
Bases: *blocks.bricks.cost.Cost*

apply

class blocks.bricks.cost.**Cost**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

apply

class blocks.bricks.cost.**CostMatrix**(*name=None*, *children=None*)
Bases: *blocks.bricks.cost.Cost*

Base class for costs which can be calculated element-wise.

Assumes that the data has format (batch, features).

apply

cost_matrix

class blocks.bricks.cost.**MisclassificationRate**(*top_k=1*)
Bases: *blocks.bricks.cost.Cost*

Calculates the misclassification rate for a mini-batch.

Parameters **top_k** (*int, optional*) – If the ground truth class is within the *top_k* highest responses for a given example, the model is considered to have predicted correctly. Default: 1.

Notes

Ties for *top_k*-th place are broken pessimistically, i.e. in the (in practice, rare) case that there is a tie for *top_k*-th highest output for a given example, it is considered an incorrect prediction.

apply

class blocks.bricks.cost.**SquaredError**(*name=None*, *children=None*)
Bases: *blocks.bricks.cost.CostMatrix*

cost_matrix

Wrapper bricks

class blocks.bricks.wrappers.**BrickWrapper**
Bases: object

Base class for wrapper metaclasses.

Sometimes one wants to extend a brick with the capability to handle inputs different from what it was designed to handle. A typical example are inputs with more dimensions that was foreseen at the development stage. One way to proceed in such a situation is to write a decorator that wraps all application methods of the brick class by some additional logic before and after the application call. *BrickWrapper* serves as a convenient base class for such decorators.

Note, that since directly applying a decorator to a Brick subclass will only take place after __new__() is called, subclasses of *BrickWrapper* should be applied by setting the *decorators* attribute of the new brick class, like in the example below:

```
>>> from blocks.bricks.base import Brick
>>> class WrappedBrick(Brick):
...     decorators = [WithExtraDims()]
```

wrap(*wrapped*, *namespace*)
Wrap an application of the base brick.

This method should be overriden to write into its *namespace* argument all required changes.

Parameters

- **mcs** (*type*) – The metaclass.
- **wrapped** (*Application*) – The application to be wrapped.
- **namespace** (*dict*) – The namespace of the class being created.

class blocks.bricks.wrappers.**WithExtraDims**
Bases: *blocks.bricks.wrappers.BrickWrapper*

Wraps a brick's applications to handle inputs with extra dimensions.

A brick can be often reused even when data has more dimensions than in the default setting. An example is a situation when one wants to apply *categorical_cross_entropy()* to temporal data, that is when an additional 'time' axis is prepended to its both *x* and *y* inputs.

This wrapper adds reshapes required to use application methods of a brick with such data by merging the extra dimensions with the first non-extra one. Two key assumptions are made: that all inputs and outputs have the same number of extra dimensions and that these extra dimensions are equal throughout all inputs and outputs.

While this might be inconvinient, the wrapped brick does not try to guess the number of extra dimensions, but demands it as an argument. The considerations of simplicity and reliability motivated this design choice. Upon availability in Blocks of a mechanism to request the expected number of dimensions for an input of a brick, this can be reconsidered.

wrap(*wrapped*, *namespace*)
Wrap an application of the base brick.

This method should be overriden to write into its *namespace* argument all required changes.

Parameters

- **mcs** (*type*) – The metaclass.
- **wrapped** (*Application*) – The application to be wrapped.
- **namespace** (*dict*) – The namespace of the class being created.

2.5.3 Extensions

class blocks.extensions.**CallbackName**
Bases: str

A name of a TrainingExtension callback.

Raises

- TypeError on comparison with a string which is not a name of
- TrainingExtension callback.

class blocks.extensions.**CompositeExtension**(*sub_extensions*, *run_before_children=True*, ***kwargs*)

Bases: *blocks.extensions.SimpleExtension*

An extension that manages several other extensions.

Parameters

- **sub_extensions** (*iterable*) – An iterable collection of sub-extensions to manage.
- **run_before_children** (*bool, optional*) – Whether the container extension's own logic should be dispatched before that of the sub-extensions. If `False`, the containing extension is dispatched last. Defaults to `True`.

Notes

The main use case for this class is bundling together groups of extensions that are most commonly used in tandem, configured so as to interact with one another. Encapsulating this pattern in a single extension reduces boilerplate.

Sub-extensions are dispatched in the order specified in `sub_extensions`, on whatever triggers they are individually configured to respect.

Sub-extensions may be run on different triggers than the containing extension; the trigger keywords passed to the constructor for this class only affect the outer extension's logic, and sub-extensions should be configured independently (possibly in a constructor for a subclass of *CompositeExtension*).

dispatch(*callback_invoked*, **from_main_loop*)

Check conditions and call the *do()* method.

Also adds additional arguments if specified for a condition.

Todo: Add a check for a situation when several conditions are met at the same time and do something.

---

do(*which_callback*, **args*)

Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

main_loop

class blocks.extensions.**FinishAfter**(***kwargs*)

Bases: *blocks.extensions.SimpleExtension*

Finishes the training process when triggered.

do(*which_callback*, **args*)

Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

class blocks.extensions.**Predicate**(*condition*, *num*)
Bases: object

class blocks.extensions.**Printing**(***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Prints log messages to the screen.

do(*which_callback*, **args*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

class blocks.extensions.**ProgressBar**(***kwargs*)
Bases: *blocks.extensions.TrainingExtension*

Display a progress bar during training.

This extension tries to infer the number of iterations per epoch by querying the *num_batches*, *num_examples* and *batch_size* attributes from the `IterationScheme`. When this information is not available it will display a simplified progress bar that does not include the estimated time until the end of this epoch.

Notes

This extension should be run before other extensions that print to the screen at the end or at the beginning of the epoch (e.g. the *Printing* extension). Placing ProgressBar before these extension will ensure you won't get intermingled output on your terminal.

after_epoch()
The callback invoked after an epoch is finished.

before_batch(*batch*)
The callback invoked before a batch is processed.

Parameters **batch** (*object*) – The data batch to be processed.

before_epoch()
The callback invoked before starting an epoch.

create_bar()
Create a new progress bar.

Calls *self.get_iter_per_epoch()*, selects an appropriate set of widgets and creates a ProgressBar.

get_iter_per_epoch()
Try to infer the number of iterations per epoch.

class blocks.extensions.**SimpleExtension**(***kwargs*)
Bases: *blocks.extensions.TrainingExtension*

A base class for simple extensions.

All logic of simple extensions is concentrated in the method *do()*. This method is called when certain conditions are fulfilled. The user can manage the conditions by calling the *add_condition* method and by passing arguments to the constructor. In addition to specifying when *do()* is called, it is possible to specify additional arguments passed to *do()* under different conditions.

Parameters

- **before_training** (*bool*) – If `True`, *do()* is invoked before training.
- **before_first_epoch** (*bool*) – If `True`, *do()* is invoked before the first epoch.
- **before_epoch** (*bool*) – If `True`, *do()* is invoked before every epoch.
- **on_resumption** (*bool, optional*) – If `True`, *do()* is invoked when training is resumed.
- **on_interrupt** (*bool, optional*) – If `True`, *do()* is invoked when training is interrupted.
- **after_epoch** (*bool*) – If `True`, *do()* is invoked after every epoch.
- **after_batch** (*bool*) – If `True`, *do()* is invoked after every batch.
- **after_training** (*bool*) – If `True`, *do()* is invoked after training.
- **after_n_epochs** (*int, optional*) – If not `None`, *do()* is invoked when *after_n_epochs* epochs are done.
- **every_n_epochs** (*int, optional*) – If not `None`, *do()* is invoked after every n-th epoch.
- **after_n_batches** (*int, optional*) – If not `None`, *do()* is invoked when *after_n_batches* batches are processed.
- **every_n_batches** (*int, optional*) – If not `None`, *do()* is invoked after every n-th batch.

BOOLEAN_TRIGGERS = frozenset(['before_batch', 'after_batch', 'after_training', 'before_

INTEGER_TRIGGERS = frozenset(['every_n_batches', 'after_n_epochs', 'every_n_epochs', '

add_condition(*callbacks_names*, *predicate=None*, *arguments=None*)
Adds a condition under which a *do()* is called.

Parameters

- **callbacks_names** (*list of str*) – The names of the callback in which the method.

- **predicate** (*function*) – A predicate function the main loop's log as the single parameter and returning `True` when the method should be called and `False` when should not. If `None`, an always `True` predicate is used.
- **arguments** (*iterable*) – Additional arguments to be passed to *do()*. They will be concatenated with the ones passed from the main loop (e.g. the batch in case of *after_epoch* callback).

Returns

Return type The extension object (allow chaining calls)

dispatch(*callback_invoked*, **from_main_loop*)
Check conditions and call the *do()* method.

Also adds additional arguments if specified for a condition.

Todo: Add a check for a situation when several conditions are met at the same time and do something.

do(*which_callback*, **args*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

static parse_args(*which_callback*, *args*)
Separates *do()* arguments coming from different sources.

When a *do()* method receives arguments from both the main loop (e.g. a batch) and the user, it often has to separate them. This method is the right tool to use.

Parameters

- **which_callback** (*str*) – The name of the callback.
- **args** (*iterable*) – The arguments.

Returns

- **from_main_loop** (*tuple*)
- **from_user** (*tuple*)

set_conditions(***kwargs*)
Set the conditions for which this extension should be run.

:param See the *SimpleExtension* docstring for a list of: :param possible parameters.:

class blocks.extensions.**Timestamp**(*log_record='timestamp'*, *separator=' '*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Adds a human readable (ISO 8601) timestamp to the log.

Parameters

- **log_record** (*str, optional*) – The record name to use. Defaults to 'timestamp'.
- **separator** (*str, optional*) – Separator between the date and time. ISO 8601 specifies 'T'. Here, we default to ' ' (blank space) for human readability.

Notes

By default, triggers after every epoch as well as before training starts, after training finishes, when an error occurs or when training is interrupted or resumed, as these are all generally useful circumstances for which to have a timestamp. These can be disabled by passing *False* as the appropriate keyword argument; see *SimpleExtension*.

DEFAULT_LOG_RECORD = 'timestamp'

do(**args*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

get_timestamp()

class blocks.extensions.**Timing**(*prefix=''*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Add timing information to the log.

This adds data about the time spent in the algorithm's `process_batch()` method as well as the time spent reading data per batch or epoch. It also reports the time spent initializing the algorithm.

Parameters **prefix** (*str*) – Prefix to be added to the log record. Defaults to the empty string.

Notes

Add this extension *before* the *Printing* extension.

Created with callbacks like `every_n_batches` this extension averages the time.

This extension does *not* enable full profiling information. To see a full profile of the main loop at the end of training, use the `profile` configuration (e.g. by setting `BLOCKS_PROFILE=true`).

do(*which_callback*, **args*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

class blocks.extensions.**TrainingExtension**(*name=None*)

Bases: object

The base class for training extensions.

An extension is a set of callbacks sharing a joint context that are invoked at certain stages of the training procedure. These callbacks typically add a certain functionality to the training procedure, e.g. running validation on auxiliary datasets or early stopping.

Parameters **name** (*str, optional*) – The name of the extension. The names are useful in order to distinguish between several extensions of the same type that belongs to the same main loop. By default the name is set to the name of the class.

main_loop

MainLoop – The main loop to which the extension belongs.

name

str – The name of the extension.

after_batch(*batch*)

The callback invoked after a batch is processed.

Parameters **batch** (*object*) – The data batch just processed.

after_epoch()

The callback invoked after an epoch is finished.

after_training()

The callback invoked after training is finished.

before_batch(*batch*)

The callback invoked before a batch is processed.

Parameters **batch** (*object*) – The data batch to be processed.

before_epoch()

The callback invoked before starting an epoch.

before_training()

The callback invoked before training is started.

dispatch(*callback_name*, **args*)

Runs callback with the given name.

The reason for having this method is to allow the descendants of the *TrainingExtension* to intercept callback invocations and do something with them, e.g. block when certain condition does not hold. The default implementation simply invokes the callback by its name.

main_loop

on_error(*exception*)
The callback invoked when an error occurs.

Parameters **exception** (*object*) – Exception occurred during the main loop run.

on_interrupt()
The callback invoked when training is interrupted.

on_resumption()
The callback invoked after training is resumed.

blocks.extensions.**always_true**(*log*)

blocks.extensions.**callback**(*func*)

blocks.extensions.**has_done_epochs**(*log*)

Monitoring extensions

class blocks.extensions.monitoring.**DataStreamMonitoring**(*variables*, *data_stream*, *updates=None*, ***kwargs*)

Bases: *blocks.extensions.SimpleExtension*, *blocks.extensions.monitoring.MonitoringExtension*

Monitors Theano variables and monitored-quantities on a data stream.

By default monitoring is done before the first and after every epoch.

Parameters

- **variables** (list of `TensorVariable` and) – `MonitoredQuantity` The variables to monitor. The variable names are used as record names in the logs.
- **updates** (list of tuples or `OrderedDict` or None) – `TensorSharedVariable` updates to be performed during evaluation. This parameter is only for Theano variables. Be careful not to update any model parameters as this is not intended to alter your model in any meaningful way. A typical use case of this option arises when the theano function used for evaluation contains a call to `scan()` which might have returned shared variable updates.
- **data_stream** (instance of `DataStream`) – The data stream to monitor on. A data epoch is requested each time monitoring is done.

do(*callback_name*, **args*)
Write the values of monitored variables to the log.

class blocks.extensions.monitoring.**MonitoringExtension**(*prefix=None*, *suffix=None*, ***kwargs*)

Bases: *blocks.extensions.TrainingExtension*

A mixin with logic shared by monitoring extensions.

Parameters

- **prefix** (*str, optional*) – The prefix for the log records done by the extension. It is prepended to the variable names with an underscore as a separator. If not given, no prefix is added to the names of the observed variables.
- **suffix** (*str, optional*) – The suffix for the log records done by the extension. It is appended to the end of variable names with an underscore as a separator. If not given, no suffix is added the names of the observed variables.

SEPARATOR = '_'

add_records(*log*, *record_tuples*)
Helper function to add monitoring records to the log.

record_name(*variable*)
The record name for a variable.

class blocks.extensions.monitoring.**TrainingDataMonitoring**(*variables*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*, *blocks.extensions.monitoring.MonitoringExtension*

Monitors values of Theano variables on training batches.

Use this extension to monitor a quantity on every training batch cheaply. It integrates with the training algorithm in order to avoid recomputing same things several times. For instance, if you are training a network and you want to log the norm of the gradient on every batch, the backpropagation will only be done once. By controlling the frequency with which the *do()* method is called, you can aggregate the monitored variables, e.g. only log the gradient norm average over an epoch.

Parameters **variables** (list of TensorVariable or) – MonitoredQuantity The variables or non-Theano quantities to monitor. The variable names are used as record names in the logs.

Notes

All the monitored variables are evaluated _before_ the parameter update.

Requires the training algorithm to be an instance of *UpdatesAlgorithm*.

do(*callback_name*, **args*)
Initializes the buffer or commits the values to the log.

What this method does depends on from what callback it is called and with which arguments. When called within *before_training*, it initializes the aggregation buffer and instructs the training algorithm what additional computations should be carried at each step by adding corresponding updates to it. In most_other cases it writes aggregated values of the monitored variables to the log. An exception is when an argument *just_aggregate* is given: in this cases it updates the values of monitored non-Theano quantities, but does not write anything to the log.

blocks.extensions.monitoring.**take_last**(*variable*)

Training

class blocks.extensions.training.**SharedVariableModifier**(*parameter*, *function*, *num_args=None*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Adjusts shared variable parameter using some function.

Applies a function to compute the new value of a shared parameter each iteration.

This class can be used to adapt over the training process parameters like learning rate, momentum, etc.

Parameters

- **parameter** (TensorSharedVariable) – Shared variable to be adjusted
- **function** (*callable*) – A function which outputs a numeric value to which the given shared variable will be set and may take one or two arguments.

In the first case, function that takes the total number of iterations done (`int`) as an input.

In the second case, it is a function which takes number of iterations done (`int`) and old value of the shared variable (with the same dtype as *parameter*).

- **num_args** (*int, optional*) – The number of arguments to pass to the function. If unspecified, it will be inferred. This is useful if you are using function-like objects for which the arity of the function cannot be inferred.

Notes

This class includes a method `function` that calls the function passed in the constructor and a `num_args` property which computes the number of arguments to use by inspecting the function object. Subclasses may override a method called `function` and/or the `num_args` property and instead pass `None` to the superclass constructor. This can be used to bypass certain serialization issues on Legacy Python regarding the unpicklability of instance method objects.

do(*which_callback*, **args*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

function(**args*)

num_args

class blocks.extensions.training.**TrackTheBest**(*record_name*, *notification_name=None*, *choose_best=<built-in function min>*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Check if a log quantity has the minimum/maximum value so far.

Parameters

- **record_name** (*str*) – The name of the record to track.
- **notification_name** (*str, optional*) – The name for the record to be made in the log when the current value of the tracked quantity is the best so far. It not given, 'record_name' plus "best_so_far" suffix is used.
- **choose_best** (*callable, optional*) – A function that takes the current value and the best so far and return the best of two. By default `min()`, which corresponds to tracking the minimum value.

best_name
str – The name of the status record to keep the best value so far.

notification_name
str – The name of the record written to the log when the current value of the tracked quantity is the best so far.

Notes

In the likely case that you are relying on another extension to add the tracked quantity to the log, make sure to place this extension *after* the extension that writes the quantity to the log in the *extensions* argument to `blocks.main_loop.MainLoop`.

do(*which_callback*, **args*)
Does the job of the training extension.

Parameters

- **which_callback** (`str`) – The name of the callback in the context of which `do()` is run.
- ***args** (`tuple`) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

Serialization

class blocks.extensions.saveload.**Checkpoint**(*path*, *parameters=None*, *save_separately=None*, *save_main_loop=True*, *use_cpickle=False*, ***kwargs*)
Bases: `blocks.extensions.SimpleExtension`

Saves a pickled version of the main loop to the disk.

The pickled main loop can be later reloaded and training can be resumed.

Makes a *SAVED_TO* record in the log with the serialization destination in the case of success and `None` in the case of failure. The value of the record is a tuple of paths to which saving was done (there can be more than one if the user added a condition with an argument, see `do()` docs).

Parameters

- **path** (`str`) – The destination path for pickling.
- **parameters** (`list, optional`) – The parameters to save separately. If None, the parameters from the model (main_loop.model.parameters) are saved.
- **save_separately** (`list of str, optional`) – The list of the main loop's attributes to be saved (copied) in a separate file in the tar archive. It may be used for example to save the log separetely. The name of the attribute will be used as name in the tar file.
- **save_main_loop** (`bool`) – Choose whether to save the main loop or not. This can be useful for example if you are only interested in saving the parameters, but not the whole main loop. Defaults to *True*.
- **use_cpickle** (`bool`) – See documentation of `dump()`.

Notes

Using pickling for saving the whole main loop object comes with certain limitations:

- Theano computation graphs build in the GPU-mode (*theano.config.device == "gpu"*) can not be used in the usual mode (and vice-versa). Therefore using this extension binds you to using only one kind of device.

do(*callback_name*, **args*)
Pickle the main loop object to the disk.

If **args* contain an argument from user, it is treated as saving path to be used instead of the one given at the construction stage.

class blocks.extensions.saveload.**Load**(*path*, *load_iteration_state=False*, *load_log=False*, ***kwargs*)
Bases: *blocks.extensions.SimpleExtension*

Loads a saved checkpoint into the main loop.

Makes a *LOADED_FROM* record in the log with the dump path.

Parameters

- **path** (*str*) – The path to the folder with dump.
- **load_iteration_state** (*bool*) – If *True*, load the iteration state. This can be useful when your model has very long epochs, and you want to resume when you were in the middle of one. Defaults to *False*.
- **load_log** (*bool*) – If *True*, load the old log and continue logging from there. Convenient because you end up with a single log of the entire training history. Defaults to *False*.

Notes

Requires the model to be created entirely using bricks, with a unique path/name for each brick, so that the parameters can be matched to their values.

In order to load the iteration state and the log, the saved model needs to be unpickled. Note that resuming training this way is still not entirely seamless because e.g. extensions will not be reloaded.

do(**args*, ***kwargs*)
Does the job of the training extension.

Parameters

- **which_callback** (*str*) – The name of the callback in the context of which *do()* is run.
- ***args** (*tuple*) – The arguments from the main loop concatenated with additional arguments from user.

Notes

Subclasses *must* accept additional positional arguments in their call signature for this method, even if they are unused.

load_to(*main_loop*)

2.5.4 Filter

class blocks.filter.**VariableFilter**(*roles=None*, *bricks=None*, *each_role=False*, *name=None*, *name_regex=None*, *theano_name=None*, *theano_name_regex=None*, *call_id=None*, *applications=None*)

Bases: object

Filters Theano variables based on a range of criteria.

Parameters

- **roles** (list of *VariableRole* instances, optional) – Matches any variable which has one of the roles given.
- **bricks** (list of *Brick* classes or list of) – instances of *Brick*, optional Matches any variable that is instance of any of the given classes or that is owned by any of the given brick instances.
- **each_role** (*bool, optional*) – If `True`, the variable needs to have all given roles. If `False`, a variable matching any of the roles given will be returned. `False` by default.
- **name** (*str, optional*) – The variable name. The Blocks name (i.e. *x.tag.name*) is used.
- **name_regex** (*str, optional*) – A regular expression for the variable name. The Blocks name (i.e. *x.tag.name*) is used.
- **theano_name** (*str, optional*) – The variable name. The Theano name (i.e. *x.name*) is used.
- **theano_name_regex** (*str, optional*) – A regular expression for the variable name. The Theano name (i.e. *x.name*) is used.
- **call_id** (*str, optional*) – The call identifier as written in *ApplicationCall* metadata attribute.
- **applications** (list of *Application*) – or *BoundApplication*, optional Matches a variable that was produced by any of the applications given.

Notes

Note that only auxiliary variables, parameters, inputs and outputs are tagged with the brick that created them. Other Theano variables that were created in the process of applying a brick will be filtered out.

Note that technically speaking, bricks are able to have non-shared variables as parameters. For example, we can use the transpose of another weight matrix as the parameter of a particular brick. This means that in some unusual cases, filtering by the `PARAMETER` role alone will not be enough to retrieve all trainable parameters in your model; you will need to filter out the shared variables from these (using e.g. `is_shared_variable()`).

Examples

```
>>> from blocks.bricks import MLP, Linear, Logistic, Identity
>>> from blocks.roles import BIAS
>>> mlp = MLP(activations=[Identity(), Logistic()], dims=[20, 10, 20])
>>> from theano import tensor
>>> x = tensor.matrix()
>>> y_hat = mlp.apply(x)
```

(continues on next page)

(continued from previous page)

```
>>> from blocks.graph import ComputationGraph
>>> cg = ComputationGraph(y_hat)
>>> from blocks.filter import VariableFilter
>>> var_filter = VariableFilter(roles=[BIAS],
...                             bricks=[mlp.linear_transformations[0]])
>>> var_filter(cg.variables)
[b]
```

blocks.filter.**get_annotation**(*var*, *cls*)
A helper function to retrieve an annotation of a particular type.

Notes

This function returns the first annotation of a particular type. If there are multiple–there shouldn't be–it will ignore them.

blocks.filter.**get_application_call**(*var*)
Retrieves the application call that created this variable.

See *get_annotation()*.

blocks.filter.**get_brick**(*var*)
Retrieves the brick that created this variable.

See *get_annotation()*.

2.5.5 Computational graph

class blocks.graph.**ComputationGraph**(*outputs*)
Bases: object

Encapsulates a managed Theano computation graph.

This implies that it not only contains the variables required to compute the given outputs, but also all the auxiliary variables and updates that were attached to these variables through the annotation system.

All variables are presented in topologically sorted order according to the apply nodes that they are an input to.

Parameters **outputs** ((list of) TensorVariable) – The output(s) of the computation graph.

inputs
list of TensorVariable – The inputs of the computation graph. This does not include shared variables and constants.

shared_variables
list of TensorSharedVariable – All the shared variables in the graph.

parameters
list of TensorSharedVariable – All the shared variables which have the PARAMETER role.

outputs
list of TensorVariable – The outputs of the computations graph (as passed to the constructor).

auxiliary_variables
list of TensorVariable – All variables which have the AUXILIARY role.

intermediary_variables
list of TensorVariable – Any variable that is not part of *inputs* or *outputs*.

variables
list of `TensorVariable` – All variables (including auxiliary) in the managed graph.

scans
list of `Scan` – All Scan ops used in this computation graph.

scan_variables
list of `TensorVariable` – All variables of the inner graphs of Scan ops.

updates
`TensorSharedVariable` updates – All the updates found attached to the annotations.

auxiliary_variables

dict_of_inputs()
Return a mapping from an input name to the input.

get_snapshot(*data*)
Evaluate all role-carrying Theano variables on given data.

Parameters **data** (*dict of (data source, data) pairs*) – Data for input variables. The sources should match with the names of the input variables.

Returns

Return type Dictionary of (variable, variable value on given data) pairs.

get_theano_function(*additional_updates=None*, ***kwargs*)
Create Theano function from the graph contained.

Parameters ****kwargs** (*dict*) – Keyword arguments to theano.function. Useful for specifying compilation modes or profiling.

has_inputs(*variable*)
Check if a variable depends on input variables.

Returns `True` if the given variable depends on input variables, `False` otherwise.

Return type bool

inputs
Inputs to the graph, excluding constants and shared variables.

intermediary_variables

parameters

replace(*replacements*)
Replace certain variables in the computation graph.

Parameters **replacements** (*dict*) – The mapping from variables to be replaced to the corresponding substitutes.

Examples

```
>>> import theano
>>> from theano import tensor, function
>>> x = tensor.scalar('x')
>>> y = x + 2
>>> z = y + 3
>>> a = z + 5
```

Let's suppose we have dependent replacements like

```
>>> replacements = {y: x * 2, z: y * 3}
>>> cg = ComputationGraph([a])
>>> theano.pprint(a)
'(((x + TensorConstant{2}) + TensorConstant{3}) +
TensorConstant{5})'
>>> cg_new = cg.replace(replacements)
>>> theano.pprint(
...     cg_new.outputs[0])
'(((x * TensorConstant{2}) * TensorConstant{3}) +
TensorConstant{5})'
```

First two sums turned into multiplications

```
>>> float(function(cg_new.inputs, cg_new.outputs)(3.)[0])
23.0
```

scan_variables
Variables of Scan ops.

shared_variables

blocks.graph.**apply_dropout**(*computation_graph*, *variables*, *drop_prob*, *rng=None*, *seed=None*, *custom_divisor=None*)
Apply dropout to specified variables in a graph.

Parameters

- **computation_graph** (instance of *ComputationGraph*) – The computation graph.
- **variables** (list of TensorVariable) – Variables to be dropped out.
- **drop_prob** (*float*) – Probability of dropping out. If you want to apply the dropout with different probabilities for different layers, call it several times.
- **rng** (MRG_RandomStreams) – Random number generator.
- **seed** (*int*) – Random seed to be used if *rng* was not specified.
- **custom_divisor** (*float or None, optional*) – Divide dropped variables by a given scalar value. If *None*, (default) dropped variables will be divided by *(1 - drop_prob)* which is equivalent to scaling by *(1 - drop_prob)* at test time as recommended in *[DROPOUT]*.

Returns **dropped_computation_graph** – A new computation graph with dropout applied to the specified variables. In order to train with, or monitor, the outputs of the original computation graph with dropout applies, use the variables contained in *dropped_computation_graph.outputs*.

Return type instance of *ComputationGraph*

Notes

For more information, see *[DROPOUT]*.

Examples

```
>>> import numpy
>>> from theano import tensor, function
>>> from blocks.bricks import MLP, Identity
>>> from blocks.filter import VariableFilter
>>> from blocks.initialization import Constant
>>> from blocks.roles import INPUT
>>> linear = MLP([Identity(), Identity()], [2, 10, 2],
...              weights_init=Constant(1), biases_init=Constant(2))
>>> x = tensor.matrix('x')
>>> y = linear.apply(x)
>>> cg = ComputationGraph(y)
```

We are going to drop out all the input variables

```
>>> inputs = VariableFilter(roles=[INPUT])(cg.variables)
```

Here we apply dropout with default setting to our computation graph

```
>>> cg_dropout = apply_dropout(cg, inputs, 0.5)
```

Dropped out variables have role *DROPOUT* and are tagged with *replacement_of* tag. Let's filter these variables and check if they have the links to original ones.

```
>>> dropped_out = VariableFilter(roles=[DROPOUT])(cg_dropout.variables)
>>> inputs_referenced = [var.tag.replacement_of for var in dropped_out]
>>> set(inputs) == set(inputs_referenced)
True
```

Compiling theano functions to forward propagate in original and dropped out graphs

```
>>> fprop = function(cg.inputs, cg.outputs[0])
>>> fprop_dropout = function(cg_dropout.inputs, cg_dropout.outputs[0])
```

Initialize an MLP and apply these functions

```
>>> linear.initialize()
>>> fprop(numpy.ones((3, 2),
...       dtype=theano.config.floatX))
array([[ 42.,  42.],
       [ 42.,  42.],
       [ 42.,  42.]]...
>>> fprop_dropout(numpy.ones((3, 2),
...               dtype=theano.config.floatX))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]]...
```

And after the second run answer is different

```
>>> fprop_dropout(numpy.ones((3, 2),
...               dtype=theano.config.floatX))
array([[   0.,   52.],
       [ 100.,    0.],
       [   0.,    0.]]...
```

blocks.graph.**apply_noise**(*computation_graph*, *variables*, *level*, *seed=None*)

Add Gaussian noise to certain variable of a computation graph.

Parameters

- **computation_graph** (instance of *ComputationGraph*) – The computation graph.
- **variables** (TensorVariable) – Variables to add noise to.
- **level** (*float*) – Noise level.
- **seed** (*int, optional*) – The seed with which MRG_RandomStreams is initialized, is set to 1 by default.

blocks.graph.**collect_parameters**(*computation_graph*, *parameters*)

Replace parameters with a single shared variable.

This can be useful if you need to calculate the full Hessian of a computational graph. It replaces parameters with slices of a single large vectors like

```
>>> from blocks.utils import shared_floatx
>>> W1 = shared_floatx(numpy.random.rand(10, 10))
>>> W2 = shared_floatx(numpy.random.rand(10, 10))
>>> all_parameters = shared_floatx(numpy.concatenate(
...     [W1.get_value().flatten(), W2.get_value().flatten()]))
>>> W1 = all_parameters[:W1.size]
>>> W2 = all_parameters[W1.size:]
```

Parameters

- **computation_graph** (*ComputationGraph* instance) – The managed Theano graph in which to collect parameters.
- **parameters** (*list of Theano shared variables*) – The parameters whose values should be collected.

Returns A new Theano graph which has all the given parameters collected into a single large shared variable.

Return type ComputationGraph instance

Notes

Note that this replacement makes the training of the model significantly slower because of the large amount of Theano's set_subtensor calls needed to train the model.

Examples

```
>>> from blocks.bricks import MLP, Logistic
>>> from blocks.bricks.cost import SquaredError
>>> from theano import tensor
>>> x = tensor.matrix()
>>> mlp = MLP(activations=[Logistic(), Logistic()],
...           dims=[784, 100, 784])
>>> cost = SquaredError().apply(x, mlp.apply(x))
>>> cg = ComputationGraph(cost)
>>> new_cg = collect_parameters(cg, cg.shared_variables)
```

The new graph only has a single shared variable. This variable receives the COLLECTOR role.

```
>>> new_cg.shared_variables
[collected_parameters]
```

The bricks' variables have been replaced with reshaped segments of this single shared variable. These replacements are given the `COLLECTED` role.

```
>>> from blocks.filter import VariableFilter
>>> from blocks.roles import PARAMETER
>>> var_filter = VariableFilter(roles=[COLLECTED])
>>> var_filter(new_cg.variables)
[Reshape{1}.0, Reshape{1}.0, Reshape{2}.0, Reshape{2}.0]
```

2.5.6 Parameter initialization

class blocks.initialization.**Constant**(*constant*)

Bases: *blocks.initialization.NdarrayInitialization*

Initialize parameters to a constant.

The constant may be a scalar or a `ndarray` of any shape that is broadcastable with the requested parameter arrays.

Parameters **constant** (`ndarray`) – The initialization value to use. Must be a scalar or an ndarray (or compatible object, such as a nested list) that has a shape that is broadcastable with any shape requested by *initialize*.

generate(*rng*, *shape*)

Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (`numpy.random.RandomState`) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype `config.floatX`.

Return type `ndarray`

class blocks.initialization.**Identity**(*mult=1*)

Bases: *blocks.initialization.NdarrayInitialization*

Initialize to the identity matrix.

Only works for 2D arrays. If the number of columns is not equal to the number of rows, the array will be truncated or padded with zeros.

Parameters **mult** (*float, optional*) – Multiply the identity matrix with a scalar. Defaults to 1.

generate(*rng*, *shape*)

Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (`numpy.random.RandomState`) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype `config.floatX`.

Return type ndarray

class blocks.initialization.**IsotropicGaussian**(*std=1*, *mean=0*)
Bases: *blocks.initialization.NdarrayInitialization*

Initialize parameters from an isotropic Gaussian distribution.

Parameters

- **std** (*float, optional*) – The standard deviation of the Gaussian distribution. Defaults to 1.
- **mean** (*float, optional*) – The mean of the Gaussian distribution. Defaults to 0

Notes

Be careful: the standard deviation goes first and the mean goes second!

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

class blocks.initialization.**NdarrayInitialization**
Bases: object

Base class specifying the interface for ndarray initialization.

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

initialize(*var*, *rng*, *shape=None*)
Initialize a shared variable with generated parameters.

Parameters

- **var** (*object*) – A Theano shared variable whose value will be set with values drawn from this *NdarrayInitialization* instance.
- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

class blocks.initialization.**Orthogonal**(*scale=1*)
Bases: *blocks.initialization.NdarrayInitialization*

Initialize a random orthogonal matrix.

Only works for 2D arrays.

Parameters **scale** (*float, optional*) – Multiply the resulting matrix with a scalar. Defaults to 1. For a discussion of the importance of scale for training time and generalization refer to *[Saxe2013]*.

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

class blocks.initialization.**Sparse**(*num_init*, *weights_init*, *sparse_init=None*)
Bases: *blocks.initialization.NdarrayInitialization*

Initialize only a fraction of the weights, row-wise.

Parameters

- **num_init** (*int or float*) – If int, this is the number of weights to initialize per row. If float, it's the fraction of the weights per row to initialize.
- **weights_init** (*NdarrayInitialization* instance) – The initialization scheme to initialize the weights with.
- **sparse_init** (*NdarrayInitialization* instance, optional) – What to set the non-initialized weights to (0. by default)

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

class blocks.initialization.**SparseND**(*axis*, ***kwargs*)
Bases: *blocks.initialization.Sparse*

Initialize only a fraction of the weights with configurable axes.

Parameters **axis** (*int or sequence*) – Which axis or axes are to be treated as a "unit" for the purpose of the number of elements initialized. For example, an axis of (0, 1) when initializing a 4D tensor *W* will treat the first two axes of the weight tensor as a grid and initialize *num_init* elements of *W[0, 0, :, :]*, another *num_init* elements of *W[0, 1, :, :]*, and so on.

Notes

See *Sparse* for documentation of other arguments.

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

class blocks.initialization.**Uniform**(*mean=0.0*, *width=None*, *std=None*)
Bases: *blocks.initialization.NdarrayInitialization*

Initialize parameters from a uniform distribution.

Parameters

- **mean** (*float, optional*) – The mean of the uniform distribution (i.e. the center of mass for the density function); Defaults to 0.
- **width** (*float, optional*) – One way of specifying the range of the uniform distribution. The support will be [mean - width/2, mean + width/2]. **Exactly one** of *width* or *std* must be specified.
- **std** (*float, optional*) – An alternative method of specifying the range of the uniform distribution. Chooses the width of the uniform such that random variates will have a desired standard deviation. **Exactly one** of *width* or *std* must be specified.

generate(*rng*, *shape*)
Generate an initial set of parameters from a given distribution.

Parameters

- **rng** (numpy.random.RandomState) –
- **shape** (*tuple*) – A shape tuple for the requested parameter array shape.

Returns **output** – An ndarray with values drawn from the distribution specified by this object, of shape *shape*, with dtype config.floatX.

Return type ndarray

2.5.7 Logging

Log has two different backends configurable in `.blocksrc`, see *Configuration*.

Dictionary backend

class blocks.log.log.**TrainingLog**
Bases: collections.defaultdict, *blocks.log.log.TrainingLogBase*

Training log using a *defaultdict* as backend.

Notes

For analysis of the logs, it can be useful to convert the log to a Pandas data frame:

```
df = DataFrame.from_dict(log, orient='index')
```

class blocks.log.log.**TrainingLogBase**(*uuid=None*)
Bases: object

Base class for training log.

A training log stores the training timeline, statistics and other auxiliary information. Training logs can use different backends e.g. in-memory Python objects or an SQLite database.

Information is stored similar to a nested dictionary, so use `log[time][key]` to read data. An entry without stored data will return an empty dictionary-like object that can be written to, `log[time][key] = value`.

Depending on the backend, `log[time] = {'key': 'value'}` could fail. Use `log[time].update({'key': 'value'})` for compatibility across backends.

In addition to the set of records displaying training dynamics, a training log has a *status* attribute, which is a dictionary with data that is not bound to a particular time.

Warning: Changes to mutable objects might not be reflected in the log, depending on the backend. So don't use `log.status['key'].append(...)`, use `log.status['key'] = ...` instead.

Parameters **uuid** (uuid.UUID, optional) – The UUID of this log. For persistent log backends, passing the UUID will result in an old log being loaded. Otherwise a new, random UUID will be created.

status
dict – A dictionary with data representing the current state of training. By default it contains `iterations_done`, `epochs_done` and `_epoch_ends` (a list of time stamps when epochs ended).

current_row

h_uuid
Return a hexadecimal version of the UUID bytes.

This is necessary to store ids in an SQLite database.

last_epoch_row

previous_row

resume()
Resume a log by setting a new random UUID.

Keeps a record of the old log that this is a continuation of. It copies the status of the old log into the new log.

Sqlite backend

class blocks.log.sqlite.**SQLiteEntry**(*log*, *time*)
Bases: _abcoll.MutableMapping

Store log entries in an SQLite database.

Each entry is a row with the columns *uuid*, *time* (iterations done), *key* and *value*. Note that SQLite only supports numeric values, strings, and bytes (e.g. the *uuid* column), all other objects will be pickled before being stored.

Entries are automatically retrieved from ancestral logs (i.e. logs that were resumed from).

class blocks.log.sqlite.**SQLiteLog**(*database=None*, ***kwargs*)
Bases: *blocks.log.log.TrainingLogBase*, _abcoll.Mapping

Training log using SQLite as a backend.

Parameters

- **database** (*str, optional*) – The database (file) to connect to. Can also be *:memory:*. See sqlite3.connect() for details. Uses *config.sqlite_database* by default.
- ****kwargs** – Arguments to pass to TrainingLogBase

conn

class blocks.log.sqlite.**SQLiteStatus**(*log*)
Bases: _abcoll.MutableMapping

blocks.log.sqlite.**adapt_ndarray**(*obj*)
Convert NumPy scalars to floats before storing in SQLite.

This makes it easier to inspect the database, and speeds things up.

Parameters **obj** (*ndarray*) – A NumPy array.

Returns If the array was a scalar, it returns a floating point number. Otherwise it binarizes the NumPy array using *adapt_obj()*

Return type float or memoryview

blocks.log.sqlite.**adapt_obj**(*obj*)
Binarize objects to be stored in an SQLite database.

Parameters **obj** (*object*) – Any picklable object.

Returns **blob** – A buffer (Python 2) or memoryview (Python 3) of the pickled object that can be stored as a BLOB in an SQLite database.

Return type memoryview

2.5.8 Main loop

class blocks.main_loop.**MainLoop**(*algorithm*, *data_stream*, *model=None*, *log=None*, *log_backend=None*, *extensions=None*)
Bases: object

The standard main loop of Blocks.

In the *MainLoop* a model is trained by a training algorithm using data extracted from a data stream. This process is scrupulously documented in a log object.

The *MainLoop* itself does very little: only fetching the data from the data stream and feeding it to the algorithm. It expects the extensions to do most of the job. A respective callback of every extension is called at every stage of training. The extensions should communicate between themselves and with the main loop object by means of making records in the log. For instance in order to stop the training procedure an extension can make a record *training_finish_requested=True* in the log. The main loop checks for such a record after every batch and every epoch and terminates when finds it.

The *MainLoop* also handles interruption signal SIGINT for you (e.g. the one program receives when you press Ctrl + C). It notes this event in the log and at the next iteration or epoch end the main loop will be gracefully finished, with calling all necessary extension callbacks and waiting until they finish.

Parameters

- **algorithm** (instance of *TrainingAlgorithm*) – The training algorithm.
- **data_stream** (instance of DataStream.) – The data stream. Should support AbstractDataStream interface from Fuel.
- **model** (instance of *ComputationGraph*, optional) – An annotated computation graph, typically represented by ComputationGraph or Model object. The main loop object uses the model only for optional sanity checks, it is here mainly for the main loop extensions.
- **log** (instance of *TrainingLog*, optional) – The log. When not given, a *TrainingLog* is created.
- **log_backend** (*str*) – The backend to use for the log. Currently *python* and *sqlite* are available. If not given, *config.log_backend* will be used. Ignored if *log* is passed.
- **extensions** (list of *TrainingExtension* instances) – The training extensions. Will be called in the same order as given here.

find_extension(*name*)
Find an extension with a given name.

Parameters **name** (*str*) – The name of the extension looked for.

Notes

Will crash if there no or several extension found.

iteration_state
Quick access to the (data stream, epoch iterator) pair.

model

run()
Starts the main loop.

The main loop ends when a training extension makes a *training_finish_requested* record in the log.

status
A shortcut for *self.log.status*.

exception blocks.main_loop.**TrainingFinish**
Bases: exceptions.Exception

An exception raised when a finish request is found in the log.

2.5.9 Model

A model in Blocks is simply an annotated computation graph. The class *Model* extends *blocks.graph.ComputationGraph* :class:, which is able to handle annotations and roles in general, but is deliberately made unaware of specific annotations that a Theano graph created by Blocks typically has, such as bricks and application calls. The *Model* adds this functionality. Using *Model* you can do things like query all the bricks used to build the computation graph, request "hierarchical names" of the parameters (a hierarchical name is a path-like string which in addition to the parameter's name contains names of the bricks on the path from a root brick to the brick that owns the parameters, e.g. /mlp/linear/W).

For more information, see *Model* docstring.

class blocks.model.**Model**(**args*, ***kwargs*)

Bases: *blocks.graph.ComputationGraph*

Handles annotations in Blocks-built computation graphs.

Use this class to handle your Blocks-created computation graph.

Examples

```
>>> from theano import tensor
>>> from blocks.bricks import MLP, Tanh
>>> x = tensor.matrix('x')
>>> mlp = MLP([Tanh(), Tanh()], [10, 10, 10])
>>> y = mlp.apply(x)
>>> model = Model(y)
```

With *Model* you can get access to the brick hierarchy. The brick hierarchy is defined by `children` attributes that every brick has. The bricks that are not children of other bricks are called top bricks. It is often useful to have access to top bricks of a brick hierarchy used to build a computation graph, and here is how you can do it:

```
>>> model.get_top_bricks()
[<blocks.bricks.sequences.MLP object at ...]
```

You can also get "hierarchical" names for the parameters, which encode the position of the owning brick in the brick hierarchy.

```
>>> model.get_parameter_dict()
OrderedDict([('/mlp/linear_1.b', b), ('/mlp/linear_0.b', b),
('/mlp/linear_0.W', W), ('/mlp/linear_1.W', W)])
```

check_sanity(*algorithm*)

get_parameter_dict()

Returns parameters with their hierarchical names.

The parameter names are formed from positions of their owner bricks in the bricks hierarchy. The variable names are used for the parameters that do not belong to any brick.

Returns **parameter_dict** – A dictionary of (hierarchical name, shared variable) pairs.

Return type dict

get_parameter_values()

Return the values of model parameters.

The same hierarhical names as in *get_parameter_dict()* are used to uniquely identify parameters.

Returns **parameter_values** – Dictionary of (hierarchical name, `ndarray`) pairs.

Return type OrderedDict

get_top_bricks()

Get the bricks that do not have parents.

Returns **bricks**

Return type list of *Brick*

set_parameter_values(*parameter_values*)
Set the values of model parameters.

The same hierarhical names as in *get_parameter_dict()* are used to uniquely identify parameters.

Parameters **parameter_values** (*OrderedDict*) – Dictionary of (hierarchical name, ndarray) pairs.

2.5.10 Variable roles

blocks.roles.**add_role**(*var*, *role*)
Add a role to a given Theano variable.

Parameters

- **var** (TensorVariable) – The variable to assign the new role to.
- **role** (*VariableRole* instance) –

Notes

Some roles are subroles of others (e.g. *WEIGHT* is a subrole of *PARAMETER*). This function will not add a role if a more specific role has already been added. If you need to replace a role with a parent role (e.g. replace *WEIGHT* with *PARAMETER*) you must do so manually.

Examples

```
>>> from theano import tensor
>>> W = tensor.matrix()
>>> from blocks.roles import PARAMETER, WEIGHT
>>> add_role(W, PARAMETER)
>>> print(*W.tag.roles)
PARAMETER
>>> add_role(W, WEIGHT)
>>> print(*W.tag.roles)
WEIGHT
>>> add_role(W, PARAMETER)
>>> print(*W.tag.roles)
WEIGHT
```

Roles

All roles are implemented as subclasses of VariableRole.

class blocks.roles.**VariableRole**
Base class for all variable roles.

The actual roles are instances of the different subclasses of VariableRole. They are:

blocks.roles.**INPUT = INPUT**
The input of a *Brick*

blocks.roles.**OUTPUT = OUTPUT**
The output of a *Brick*

blocks.roles.**AUXILIARY = AUXILIARY**
Variables added to the graph as annotations

blocks.roles.**COST = COST**
A scalar cost that can be used to train or regularize

blocks.roles.**PARAMETER = PARAMETER**
A parameter of the model

blocks.roles.**WEIGHT = WEIGHT**
The weight matrices of linear transformations

blocks.roles.**BIAS = BIAS**
Biases of linear transformations

blocks.roles.**FILTER = FILTER**
The filters (kernels) of a convolution operation

2.5.11 Brick selectors

class blocks.select.**Path**(*nodes*)
Bases: object

Encapsulates a path in a hierarchy of bricks.

Currently the only allowed elements of paths are names of the bricks and names of parameters. The latter can only be put in the end of the path. It is planned to support regular expressions in some way later.

Parameters **nodes** (*list or tuple of path nodes*) – The nodes of the path.

nodes
tuple – The tuple containing path nodes.

class BrickName
Bases: str

part()

class ParameterName
Bases: str

part()

parameter_separator = '.'

static parse(*string*)
Constructs a path from its string representation.

Todo: More error checking.

Parameters **string** (*str*) – String representation of the path.

separator = '/'

separator_re = <_sre.SRE_Pattern object>

class blocks.select.**Selector**(*bricks*)
Bases: object

Selection of elements of a hierarchy of bricks.

Parameters **bricks** (list of *Brick*) – The bricks of the selection.

get_parameters(*parameter_name=None*)
Returns parameters from selected bricks and their descendants.

Parameters **parameter_name** (*Path.ParameterName*, optional) – If given, only parameters with a *name* attribute equal to *parameter_name* are returned.

Returns **parameters** – A dictionary of (*path*, *parameter*) pairs, where *path* is a string representation of the path in the brick hierarchy to the parameter (i.e. the slash-delimited path to the brick that owns the parameter, followed by a dot, followed by the parameter's name), and *parameter* is the Theano variable representing the parameter.

Return type OrderedDict

Examples

```
>>> from blocks.bricks import MLP, Tanh
>>> mlp = MLP([Tanh(), Tanh(), Tanh()], [5, 7, 11, 2])
>>> mlp.allocate()
>>> selector = Selector([mlp])
>>> selector.get_parameters()
OrderedDict([('/mlp/linear_0.W', W), ('/mlp/linear_0.b', b),
('/mlp/linear_1.W', W), ('/mlp/linear_1.b', b),
('/mlp/linear_2.W', W), ('/mlp/linear_2.b', b)])
```

Or, select just the weights of the MLP by passing the parameter name *W*:

```
>>> w_select = Selector([mlp])
>>> w_select.get_parameters('W')
OrderedDict([('/mlp/linear_0.W', W), ('/mlp/linear_1.W', W),
('/mlp/linear_2.W', W)])
```

select(*path*)
Select a subset of current selection matching the path given.

> **Warning:** Current implementation is very inefficient (theoretical complexity is $O(n^3)$, where n is the number of bricks in the hierarchy). It can be sped up easily.

Parameters **path** (*Path* or str) – The path for the desired selection. If a string is given it is parsed into a path.

Returns

- *Depending on the path given, one of the following*
- * *Selector* with desired bricks.
- * list of `SharedTensorVariable`.

2.5.12 Serialization

This module provides *load()* and *dump()* functions that can serve as drop-in replacement for the respective functions from the standard `pickle` module. The main differences between them and the standard ones are:

- The dump is physically a tarball, in which the pickle is stored as '_pkl' file.

- A special file '_parameters' in the tarball can contain the data of a selected set of Theano shared variables. This data is referenced from *_pkl* using persistent id mechanism, which means that no duplication takes place. The goal here is to save the values of the parameters (this is what these shared variables are in most cases) in the most robust way possible. The actual format for '_parameters' file is the one used by `numpy.savez()`, i.e. a zip file of numpy arrays.
- More objects can be dumped in the archive using the *add_to_dump* function. If the object has the same parameters as the one already dumped, then you can avoid to dump those parameters thank to the persistent id mechanism.
- The *dump()* strives to catch situations when the user tries to pickle a function or a class not defined in the global namespace and give a meaningful warning.

If briefly, this module proposes a dumping mechanism which allows for greater robustness and persistence than standard pickling.

Examples

Consider a standard main loop (without an algorithm and a data stream for brevity)

```
>>> from theano import tensor
>>> from blocks.main_loop import MainLoop
>>> from blocks.bricks import MLP, Tanh, Softmax
>>> from blocks.model import Model
>>> mlp = MLP([Tanh(), None], [784, 10, 10])
>>> x = tensor.matrix('features')
>>> y = tensor.lmatrix('targets')
>>> cost = Softmax().categorical_cross_entropy(
...             y.flatten(), mlp.apply(tensor.flatten(x, outdim=2)))
>>> main_loop = MainLoop(None, None, model=Model(cost))
```

Let's see how the main loop is dumped by *dump()*

```
>>> from blocks.serialization import dump, load
>>> import tarfile
>>> with open('main_loop.tar', 'wb') as dst:
...     dump(main_loop, dst)
>>> tarball = tarfile.open('main_loop.tar', 'r')
>>> tarball
<tarfile.TarFile object at ...>
>>> tarball.getnames()
['_pkl']
>>> tarball.close()
```

As promised, the dump is a tarball. Since we did not ask for any additional magic, it just contains the pickled main loop in '_pkl' file.

Let's do something more interesting:

```
>>> with open('main_loop.tar', 'wb') as dst:
...     dump(main_loop, dst,
...          parameters=main_loop.model.parameters)
>>> tarball = tarfile.open('main_loop.tar', 'r')
>>> tarball.getnames()
['_parameters', '_pkl']
```

As requested by specifying the *_parameters* argument, the parameters were saved in a zip file.

```
>>> import numpy
>>> ps = numpy.load(tarball.extractfile(tarball.getmember('_parameters')))
>>> sorted(ps.keys())
['|mlp|linear_0.W', '|mlp|linear_0.b', '|mlp|linear_1.W', '|mlp|lin...]
>>> ps.close()
```

The names for parameters are chosen intelligently to reflect their position in the brick hierarchy, if they belong to bricks, and by simply using the *.name* attribute, if they do not.

The loading of the main loop as a whole still works:

```
>>> with open('main_loop.tar', 'rb') as src:
...     main_loop_loaded = load(src)
>>> main_loop_loaded
<blocks.main_loop.MainLoop object at ...>
```

Additionally, this module provides convenience routine *load_parameters()*:

```
>>> with open('main_loop.tar', 'rb') as src:
...     parameters = load_parameters(src)
>>> sorted(parameters.keys())
['/mlp/linear_0.W', '/mlp/linear_0.b', '/mlp/linear_1.W', '/mlp/line...]
```

Loading parameters saved by *dump()* with *load_parameters()* ensures that their hierarchical names are compatible with *Model* and *Selector* classes.

TODO: Add information about *add_to_dump()*.

blocks.serialization.**add_to_dump**(*object_*, *file_*, *name*, *parameters=None*, *use_cpickle=False*, *protocol=2*, ***kwargs*)

Pickles an object to an existing tar archive.

This function allows to dump more objects to an existing archive. If the object you want to dump posesses the same set of shared variables as the object already dumped, you can pass them to the *parameters* argument, which will avoid them to be serialized a second time. However, it won't work if the shared variable you pass to the *parameters* argument are not already in the archive.

Parameters

- **object** (*object*) – The object to pickle.
- **file** (*file*) – The destination for saving, opened in read-write mode (*r+*).
- **name** (*str*) – The name of the object you are dumping. It will be used as a file name in the archive. '_pkl' and '_paramters' are reserved names and can't be used.
- **parameters** (*list, optional*) – Shared variables whose internal numpy arrays should be saved separately in the *_parameters* field of the tar file. Must be a subset of the parameters already in the archive.
- **use_cpickle** (*bool*) – Use cPickle instead of pickle. Setting it to true will disable the warning message if you try to pickle objects from the main module! Be sure that you don't have the warning before turning this flag on. Default: False.
- **protocol** (*int, optional*) – The pickling protocol to use. Unlike Python's built-in pickle, the default is set to *2* instead of 0 for Python 2. The Python 3 default (level 3) is maintained.
- ****kwargs** – Keyword arguments to be passed to *pickle.Pickler*.

blocks.serialization.**continue_training**(*path*)

Continues training using checkpoint.

Parameters **path** (*str*) – Path to checkpoint.

Notes

Python picklers can unpickle objects from global namespace only if they are present in namespace where unpickling happens. Often global functions are needed for mapping, filtering and other data stream operations. In a case if the main loop uses global objects and this function fails with a message like ` `AttributeError: 'module' object has no attribute '...'` ` it means that you need to import these objects.

Examples

This function can be used in two ways: in your script where a main loop defined or in a different script. For later options see Notes section.

blocks.serialization.**dump**(*object_*, *file_*, *parameters=None*, *use_cpickle=False*, *protocol=2*, ***kwargs*)

Pickles an object, optionally saving its parameters separately.

Parameters

- **object** (*object*) – The object to pickle. If None, only the parameters passed to the *parameters* argument will be saved.
- **file** (*file*) – The destination for saving.
- **parameters** (*list, optional*) – Shared variables whose internal numpy arrays should be saved separately in the *_parameters* field of the tar file.
- **pickle_object** (*bool*) – If False, *object_* will not be serialized, only its parameters. This flag can be used when *object_* is not serializable, but one still want to save its parameters. Default: True
- **use_cpickle** (*bool*) – Use cPickle instead of pickle. Setting it to true will disable the warning message if you try to pickle objects from the main module, so be sure that there is no warning before turning this flag on. Default: False.
- **protocol** (*int, optional*) – The pickling protocol to use. Unlike Python's built-in pickle, the default is set to *2* instead of 0 for Python 2. The Python 3 default (level 3) is maintained.
- ****kwargs*** – Keyword arguments to be passed to *pickle.Pickler*.

blocks.serialization.**dump_and_add_to_dump**(*object_*, *file_*, *parameters=None*, *to_add=None*, *use_cpickle=False*, *protocol=2*, ***kwargs*)

Calls both *dump* and *add_to_dump* to serialze several objects.

This function is used to serialize several at the same time, using persistent ID. Its main advantage is that it can be used with *secure_dump*.

Parameters

- **object** (*object*) – The object to pickle. If None, only the parameters passed to the *parameters* argument will be saved.
- **file** (*file*) – The destination for saving.
- **parameters** (*list, optional*) – Shared variables whose internal numpy arrays should be saved separately in the *_parameters* field of the tar file.

- **to_add** (*dict of objects*) – A {'name': object} dictionnary of additional objects to save in the tar archive. Its keys will be used as name in the tar file.
- **use_cpickle** (*bool*) – Use cPickle instead of pickle. Setting it to true will disable the warning message if you try to pickle objects from the main module, so be sure that there is no warning before turning this flag on. Default: False.
- **protocol** (*int, optional*) – The pickling protocol to use. Unlike Python's built-in pickle, the default is set to *2* instead of 0 for Python 2. The Python 3 default (level 3) is maintained.
- ****kwargs** – Keyword arguments to be passed to *pickle.Pickler*.

blocks.serialization.**load**(*file_*, *name='_pkl'*, *use_cpickle=False*, ***kwargs*)

Loads an object saved using the *dump* function.

By default, this function loads the object saved by the *dump* function. If some objects have been added to the archive using the *add_to_dump* function, then you can load them by passing their name to the *name* parameter.

Parameters

- **file** (*file*) – The file that contains the object to load.
- **name** (*str*) – Name of the object to load. Default is *_pkl*, meaning that it is the original object which have been dumped that is loaded.
- **use_cpickle** (*bool*) – Use cPickle instead of pickle. Default: False.
- ****kwargs** – Keyword arguments to be passed to *pickle.Unpickler*. Used for e.g. specifying the encoding so as to load legacy Python pickles under Python 3.x.

Returns

Return type The object saved in `file_`.

blocks.serialization.**load_parameters**(*file_*)

Loads the parameter values saved by *dump()*.

This functions loads the parameters that have been saved separately by *dump()*, ie the ones given to its parameter *parameters*.

Parameters **file** (*file*) – The source to load the parameters from.

Returns

Return type A dictionary of (parameter name, numpy array) pairs.

blocks.serialization.**secure_dump**(*object_*, *path*, *dump_function=<function dump>*, ***kwargs*)

Robust serialization - does not corrupt your files when failed.

Parameters

- **object** (*object*) – The object to be saved to the disk.
- **path** (*str*) – The destination for saving.
- **dump_function** (*function*) – The function that is used to perform the serialization. Must take an object and file object as arguments. By default, *dump()* is used. An alternative would be `pickle.dump()`.
- ****kwargs** – Keyword arguments to be passed to *dump_function*.

2.5.13 Theano expressions

blocks.theano_expressions.**hessian_times_vector**(*gradient*, *parameter*, *vector*, *r_op=False*)

Return an expression for the Hessian times a vector.

Parameters

- **gradient** (TensorVariable) – The gradient of a cost with respect to *parameter*
- **parameter** (TensorVariable) – The parameter with respect to which to take the gradient
- **vector** (TensorVariable) – The vector with which to multiply the Hessian
- **r_op** (*bool, optional*) – Whether to use Rop() or not. Defaults to False. Which solution is fastest normally needs to be determined by profiling.

blocks.theano_expressions.**l2_norm**(*tensors*, *squared=False*)

Computes the total L2 norm of a set of tensors.

Converts all operands to TensorVariable (see as_tensor_variable()).

Parameters

- **tensors** (iterable of TensorVariable (or compatible)) – The tensors.
- **squared** (*bool, optional*) – If *True*, return the squared L2 norm. Default: *False*.

2.5.14 Common Utilities

blocks.utils.utils.**change_recursion_limit**(**args*, ***kwds*)

Temporarily changes the recursion limit.

blocks.utils.utils.**dict_subset**(*dict_*, *keys*, *pop=False*, *must_have=True*)

Return a subset of a dictionary corresponding to a set of keys.

Parameters

- **dict** (*dict*) – The dictionary.
- **keys** (*iterable*) – The keys of interest.
- **pop** (*bool*) – If True, the pairs corresponding to the keys of interest are popped from the dictionary.
- **must_have** (*bool*) – If True, a ValueError will be raised when trying to retrieve a key not present in the dictionary.

Returns **result** – An ordered dictionary of retrieved pairs. The order is the same as in the keys argument.

Return type OrderedDict

blocks.utils.utils.**dict_union**(**dicts*, ***kwargs*)

Return union of a sequence of disjoint dictionaries.

Parameters

- **dicts** (*dicts*) – A set of dictionaries with no keys in common. If the first dictionary in the sequence is an instance of *OrderedDict*, the result will be OrderedDict.
- ****kwargs** – Keywords and values to add to the resulting dictionary.

Raises ValueError – If a key appears twice in the dictionaries or keyword arguments.

blocks.utils.utils.**extract_args**(*expected*, **args*, ***kwargs*)
Route keyword and positional arguments to a list of names.

A frequent situation is that a method of the class gets to know its positional arguments only when an instance of the class has been created. In such cases the signature of such method has to be **args, **kwargs*. The downside of such signatures is that the validity of a call is not checked.

Use *extract_args()* if your method knows at runtime, but not at evaluation/compile time, what arguments it actually expects, in order to check that they are correctly received.

Parameters

- **expected** (*list of str*) – A list of strings denoting names for the expected arguments, in order.
- **args** (*iterable*) – Positional arguments that have been passed.
- **kwargs** (*Mapping*) – Keyword arguments that have been passed.

Returns **routed_args** – An OrderedDict mapping the names in *expected* to values drawn from either *args* or *kwargs* in the usual Python fashion.

Return type OrderedDict

Raises

- KeyError – If a keyword argument is passed, the key for which is not contained within *expected*.
- TypeError – If an expected argument is accounted for in both the positional and keyword arguments.
- ValueError – If certain arguments in *expected* are not assigned a value by either a positional or keyword argument.

blocks.utils.utils.**find_bricks**(*top_bricks*, *predicate*)
Walk the brick hierarchy, return bricks that satisfy a predicate.

Parameters

- **top_bricks** (*list*) – A list of root bricks to search downward from.
- **predicate** (*callable*) – A callable that returns *True* for bricks that meet the desired criteria or *False* for those that don't.

Returns **found** – A list of all bricks that are descendants of any element of *top_bricks* that satisfy *predicate*.

Return type list

blocks.utils.utils.**ipdb_breakpoint**(*x*)
A simple hook function for put_hook() that runs ipdb.

Parameters **x** (ndarray) – The value of the hooked variable.

blocks.utils.utils.**pack**(*arg*)
Pack variables into a list.

Parameters **arg** (*object*) – Either a list or tuple, or any other Python object. Lists will be returned as is, and tuples will be cast to lists. Any other variable will be returned in a singleton list.

Returns List containing the arguments

Return type list

blocks.utils.utils.**print_shape**(*x*, *header=None*)

blocks.utils.utils.**print_sum**(*x*, *header=None*)

blocks.utils.utils.**repr_attrs**(*instance*, **attrs*)

Prints a representation of an object with certain attributes.

Parameters

- **instance** (*object*) – The object of which to print the string representation
- ***attrs** – Names of attributes that should be printed.

Examples

```
>>> class A(object):
...     def __init__(self, value):
...         self.value = value
>>> a = A('a_value')
>>> repr(a)
<blocks.utils.A object at 0x7fb2b4741a10>
>>> repr_attrs(a, 'value')
<blocks.utils.A object at 0x7fb2b4741a10: value=a_value>
```

blocks.utils.utils.**reraise_as**(*new_exc*)

Reraise an exception as a different type or with a message.

This function ensures that the original traceback is kept, making for easier debugging.

Parameters **new_exc** (Exception or str) – The new error to be raised e.g. (ValueError("New message")) or a string that will be prepended to the original exception message

Notes

Note that when reraising exceptions, the arguments of the original exception are cast to strings and appended to the error message. If you want to retain the original exception arguments, please use:

```
>>> try:
...     1 / 0
... except Exception as e:
...     reraise_as(Exception("Extra information", *e.args))
Traceback (most recent call last):
  ...
Exception: 'Extra information, ...
```

Examples

```
>>> class NewException(Exception):
...     def __init__(self, message):
...         super(NewException, self).__init__(message)
>>> try:
...     do_something_crazy()
... except Exception:
...     reraise_as(NewException("Informative message"))
Traceback (most recent call last):
  ...
NewException: Informative message ...
```

blocks.utils.utils.**unpack**(*arg*, *singleton=False*)
Unpack variables from a list or tuple.

Parameters

- **arg** (*object*) – Either a list or tuple, or any other Python object. If passed a list or tuple of length one, the only element of that list will be returned. If passed a tuple of length greater than one, it will be cast to a list before returning. Any other variable will be returned as is.
- **singleton** (*bool*) – If `True`, *arg* is expected to be a singleton (a list or tuple with exactly one element) and an exception is raised if this is not the case. `False` by default.

Returns A list of length greater than one, or any other Python object except tuple.

Return type object

2.5.15 Theano Utilities

blocks.utils.theano_utils.**check_theano_variable**(*variable*, *n_dim*, *dtype_prefix*)
Check number of dimensions and dtype of a Theano variable.

If the input is not a Theano variable, it is converted to one. *None* input is handled as a special case: no checks are done.

Parameters

- **variable** (`TensorVariable` or convertible to one) – A variable to check.
- **n_dim** (*int*) – Expected number of dimensions or None. If None, no check is performed.
- **dtype_prefix** (*str*) – Expected dtype prefix or None. If None, no check is performed.

blocks.utils.theano_utils.**is_graph_input**(*variable*)
Check if variable is a user-provided graph input.

To be considered an input the variable must have no owner, and not be a constant or shared variable.

Parameters **variable** (`TensorVariable`) –

Returns `True` If the variable is a user-provided input to the graph.

Return type bool

blocks.utils.theano_utils.**is_shared_variable**(*variable*)
Check if a variable is a Theano shared variable.

Notes

This function excludes shared variables that store the state of Theano random number generators.

blocks.utils.theano_utils.**put_hook**(*variable*, *hook_fn*, **args*)
Put a hook on a Theano variables.

Ensures that the hook function is executed every time when the value of the Theano variable is available.

Parameters

- **variable** (`TensorVariable`) – The variable to put a hook on.
- **hook_fn** (*function*) – The hook function. Should take a single argument: the variable's value.
- ***args** (*list*) – Positional arguments to pass to the hook function.

blocks.utils.theano_utils.**shared_floatx**(*value*, *name=None*, *borrow=False*, *dtype=None*, ***kwargs*)

Transform a value into a shared variable of type floatX.

Parameters

- **value** (ndarray) – The value to associate with the Theano shared.
- **name** (str, optional) – The name for the shared variable. Defaults to *None*.
- **borrow** (bool, optional) – If set to True, the given *value* will not be copied if possible. This can save memory and speed. Defaults to False.
- **dtype** (str, optional) – The *dtype* of the shared variable. Default value is config.floatX.
- ****kwargs** – Keyword arguments to pass to the shared() function.

Returns A Theano shared variable with the requested value and *dtype*.

Return type tensor.TensorSharedVariable

blocks.utils.theano_utils.**shared_floatx_nans**(*shape*, ***kwargs*)

Creates a shared variable array filled with nans.

Parameters

- **shape** (*tuple*) – A tuple of integers representing the shape of the array.
- ****kwargs** – Keyword arguments to pass to the *shared_floatx()* function.

Returns A Theano shared variable filled with nans.

Return type class:'tensor.TensorSharedVariable'

blocks.utils.theano_utils.**shared_floatx_zeros**(*shape*, ***kwargs*)

Creates a shared variable array filled with zeros.

Parameters

- **shape** (*tuple*) – A tuple of integers representing the shape of the array.
- ****kwargs** – Keyword arguments to pass to the *shared_floatx()* function.

Returns A Theano shared variable filled with zeros.

Return type class:'tensor.TensorSharedVariable'

blocks.utils.theano_utils.**shared_floatx_zeros_matching**(*shared_variable*, *name=None*, ***kwargs*)

Create another shared variable with matching shape and broadcast.

Parameters

- **shared_variable** (*:class:'tensor.TensorSharedVariable'*) – A Theano shared variable with the desired shape and broadcastable flags.
- **name** (str, optional) – The name for the shared variable. Defaults to *None*.
- ****kwargs** – Keyword arguments to pass to the *shared_floatx_zeros()* function.

Returns A new shared variable, initialized to all zeros, with the same shape and broadcastable flags as *shared_variable*.

Return type class:'tensor.TensorSharedVariable'

blocks.utils.theano_utils.**shared_like**(*variable*, *name=None*, ***kwargs*)

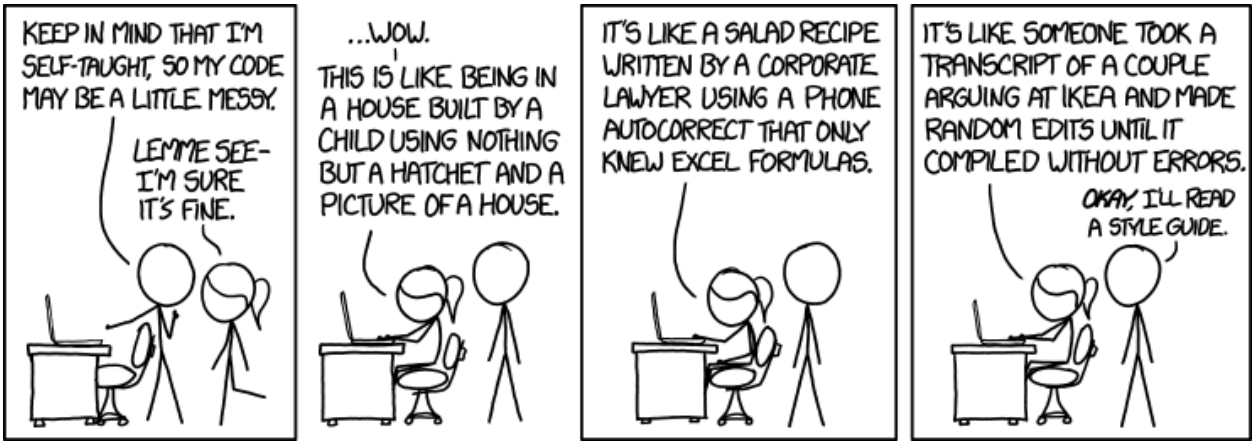
Construct a shared variable to hold the value of a tensor variable.

Parameters

- **variable** (TensorVariable) – The variable whose dtype and ndim will be used to construct the new shared variable.
- **name** (str or None) – The name of the shared variable. If None, the name is determined based on variable's name.
- ****kwargs** – Keyword arguments to pass to the shared() function.

2.6 Development

We want to encourage everyone to contribute to the development of Blocks and Fuel. To ensure the codebase is of high quality, we ask all new developers to have a quick read through these rules to make sure that any code you contribute will be easy to merge!



2.6.1 Formatting guidelines

Blocks follows the PEP8 style guide closely, so please make sure you are familiar with it. Our Travis CI buildbots (for Blocks, Fuel, and Blocks-extras) run flake8 as part of every build, which checks for PEP8 compliance (using the pep8 tool) and for some common coding errors using pyflakes. You might want to install and run flake8 on your code before submitting a PR to make sure that your build doesn't fail because of e.g. a bit of extra whitespace.

Note that passing flake8 does not necessarily mean that your code is PEP8 compliant! Some guidelines which aren't checked by flake8:

- Imports should be grouped into standard library, third party, and local imports with a blank line in between groups.
- Variable names should be explanatory and unambiguous.

There are also some style guideline decisions that were made specifically for Blocks and Fuel:

- Do not rename imports i.e. do not use `import theano.tensor as T` or `import numpy as np`.
- Direct imports, `import ...`, precede `from ... import ...` statements.
- Imports are otherwise listed alphabetically.
- Don't recycle variable names (i.e. don't use the same variable name to refer to different things in a particular part of code), especially when they are arguments to functions.

- Group trivial attribute assignments from arguments and keyword arguments together, and separate them from remaining code with a blank line. Avoid the use of implicit methods such as `self.__dict__.update(locals())`.

```
class Foo(object):
    def __init__(self, foo, bar, baz=None, **kwargs):
        super(Foo, self).__init__(**kwargs)
        if baz is None:
            baz = []

        self.foo = foo
        self.bar = bar
        self.baz = baz
```

2.6.2 Code guidelines

Some guidelines to keep in mind when coding for Blocks or Fuel. Some of these are simply preferences, others stem from particular requirements we have, e.g., in order to serialize training progress, support Python 2 and 3 simultaneously, etc.

Validating function arguments

In general, be Pythonic and rely on duck typing.

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> —James Whitcomb Riley

That is, avoid trivial checks such as

```
isinstance(var, numbers.Integral)
isinstance(var, (tuple, list))
```

in cases where any number (like a float without a fractional part or a NumPy scalar) or iterable (like a dictionary view, custom iterator) would work too.

If you need to perform some sort of input validation, don't use `assert` statements. Raise a `ValueError` instead. `assert` statements should only be used for sanity tests i.e. they *should* never be triggered, unless there is a bug in the code.

Abstract classes

If a class is an abstract base class, use Python's `abc` to mark it as such.

```
from abc import ABCMeta
from six import add_metaclass
@add_metaclass(ABCMeta)
class Abstract(object):
    pass
```

Our documentation generator (Sphinx with the autodoc extension, running on Read the Docs) doesn't recognize classes which inherit the `ABCMeta` metaclass as abstract and will try to instantiate them, causing errors when building documentation. To prevent this, make sure to always use the `add_metaclass` decorator, regardless of the parent.

Python 2 and 3

Blocks and Fuel aim to be both Python 2 and Python 3 compliant using a single code-base, without using 2to3. There are many online resources which discuss the writing of compatible code. For a quick overview see the cheatsheet from Python Charmers. For non-trivial cases, we use the six compatibility library.

Documentation should be written to be Python 3 compliant.

Reraising exceptions

When catching exceptions, use the *reraise_as()* function to reraise the exception (optionally with a new message or as a different type). Not doing so clobbers the original traceback, making it impossible to use `pdb` to debug the problems.

Serialization

To ensure the reproducibility of scientific experiments, Blocks and Fuel try to make sure that stopping and resuming training doesn't affect the final results. In order to do so it takes a radical approach, serializing the entire training state using pickle. Some things cannot be pickled, so their use should be avoided when the object will be pickled as part of the main loop:

- Lambda functions
- Iterators and generators (use picklable_itertools)
- References to methods as attributes
- Any variable that lies outside of the global namespace, e.g., nested functions
- Dynamically generated classes (possible but complicated)

Mutable types as keyword argument defaults

A common source of mysterious bugs is the use of mutable types as defaults for keyword arguments.

```
class Foo(object):
    def __init__(self, bar=[]):
        bar.append('baz')
        self.bar = bar
```

Initializing two instances of this class results in two objects sharing the same attribute `bar` with the value `['baz', 'baz']`, which is often not what was intended. Instead, use:

```
class Foo(object):
    def __init__(self, bar=None):
        if bar is None:
            bar = []
        bar.append('baz')
        self.bar = bar
```

Writing error messages

Comprehensive error messages can be a great way to inform users of what could have gone wrong. However, lengthy error messages can clutter code, and implicitly concatenated strings over multiple lines are frustrating to edit. To prevent this, use a separate triple-quoted string with escaped newlines to store the detailed explanation of your error.

Keep a terse error message directly in the code though, so that someone reading the code still knows what the error is being raised for.

```
informative_error = """

You probably passed the wrong keyword argument, which caused this error. \
Please pass `b` instead of `{value}`, and have a look at the documentation \
of the `is_b` method for details."""

def is_b(value):
    """Raises an error if the value is not 'b'."""
    if value != 'b':
        raise ValueError("wrong value" + informative_error.format(value))
    return value
```

2.6.3 Unit testing

Blocks and Fuel use unit testing to ensure that individual parts of the library behave as intended. It's also essential in ensuring that parts of the library are not broken by proposed changes. Since Blocks and Fuel were designed to be used together, it is important to make sure changes in Fuel do not break Blocks.

All new code should be accompanied by extensive unit tests. Whenever a pull request is made, the full test suite is run on Travis CI, and pull requests are not merged until all tests pass. Coverage analysis is performed using coveralls. Please make sure that at the very least your unit tests cover the core parts of your committed code. In the ideal case, all of your code should be unit tested.

If you are fixing a bug, please be sure to add a unit test to make sure that the bug does not get re-introduced later on.

The test suite can be executed locally using nose2[1].

2.6.4 Writing and building documentation

The *documentation guidelines* outline how to write documentation for Blocks and Fuel, and how to build a local copy of the documentation for testing purposes.

2.6.5 Internal API

The *development API reference* contains documentation on the internal classes that Blocks uses. If you are not planning on contributing to Blocks, have a look at the *user API reference* instead.

2.6.6 Installation

See the instructions at the bottom of the *installation instructions*.

2.6.7 Sending a pull request

See our *pull request workflow* for a refresher on the general recipe for sending a pull request to Blocks or Fuel.

[1] For all tests but the doctests, nose can also be used.

2.6.8 Making a new release

Create an initial pull request and copy the following piece of markdown code. This pull request should only change the version number. Then, create a pull request to Fuel which refers the first PR. Follow the instruction carefully and check the boxes in process.

```
- **Stage 1**: Make changes in `master`:
  - [ ] Freeze other PRs.

        After we agreed to initiate the process of releasing a new version,
        other PRs shouldn't be merged.
  - [ ] Increase the version number counter of Blocks.

        Change the version number in `blocks/__init__.py`.
  - [ ] Increase the version number counter of Fuel.

        Change the version number in `fuel/version.py`.
- **Stage 2**: After two PRs merged to Blocks and Fuel:
  - [ ] Create a pull request to merge `master` into `stable`.

        Add a link to the initial PR in order not to get lost in the numerous
        pull requests.
  - [ ] Create a pull request to Fuel.

        This will be a corresponding PR to Fuel which merges its `master` into
        `stable`. Add a link to  the initial PR.
  - [ ] Check the Travis CI build log *on both the pull requests merging
        `master` into `stable`*.

        Read carefully the Travis CI messages, check that it tests the
        right version.
  - [ ] Check the Theano version.

        The `req*.txt` should refer the last development Theano version
        which is known not to have bugs.
  - [ ] Check the Fuel version in `req*.txt` files.

        We should reference the stable version of Fuel. It can be seen
        in the Travis CI output.
  - [ ] Merge Fuel pull request.
  - [ ] Merge this pull request.
- **Stage 3**: After the PRs are merged:
  - [ ] Wait the build to pass.
  - [ ] Check documentation build at ReadTheDocs.
  - [ ] Double check that the version corresponds `__version__`.
  - [ ] Create a release of Fuel by going to the
        [releases page](https://github.com/mila-udem/fuel/releases) and
        clicking "Draft new release".
  - [ ] Create a release of Blocks by going to the
        [releases page](https://github.com/mila-udem/blocks/releases) and
        clicking "Draft new release".
```

Internal API

- *Bricks*
- *Extensions*

- *Utils*

Bricks

class blocks.bricks.base.**Application**(*application_function*)
Bases: object

An application method belonging to a particular type of brick.

The application methods of each *Brick* class are automatically replaced by an instance of *Application*. This allows us to store metadata about particular application methods (such as their in- and outputs) easily.

application
callable – The original (unbounded) application function defined on the *Brick*.

delegate_function
callable – A function that takes a *Brick* instance as an argument and returns a *BoundApplication* object to which attribute requests should be routed.

properties
dict (str, callable) – A dictionary of property getters that should be called when an attribute with the given name is requested.

instances
dict (*Brick*, *BoundApplication*) – A record of bound application instances created by the descriptor protocol.

call_stack
list of *Brick* – The call stack of brick application methods. Used to check whether the current call was made by a parent brick.

brick
type – The brick class to which this instance belongs.

Raises

- ValueError – If a brick's application method is applied by another brick which does not list the former as a child.
- ValueError – If the application method's inputs and/or outputs don't match with the function signature or the values returned (respectively).

Notes

When a *Brick* is instantiated and its application method (i.e. an instance of this class) requested, the descriptor protocol (through the `__get__()` method) automatically instantiates a *BoundApplication* class and returns this. This bound application class can be used to store application information particular to a brick instance. Any attributes unknown to the bounded application are automatically routed to the application that instantiated it.

application_function

apply(*bound_application*, **args*, ***kwargs*)

call_stack = []

delegate(*f*)
Decorator to assign a delegate application.

An application method can assign a delegate application. Whenever an attribute is not available, it will be requested from the delegate instead.

Examples

```
>>> class Foo(Brick):
...     @application(outputs=['baz'])
...     def apply(self, x):
...         return x + 1
...
...     @apply.property('inputs')
...     def apply_inputs(self):
...         return ['foo', 'bar']
>>> class Bar(Brick):
...     def __init__(self, foo):
...         self.foo = foo
...
...     @application(outputs=['foo'])
...     def apply(self, x):
...         return x + 1
...
...     @apply.delegate
...     def apply_delegate(self):
...         return self.foo.apply
>>> foo = Foo()
>>> bar = Bar(foo)
>>> bar.apply.outputs
['foo']
>>> bar.apply.inputs
['foo', 'bar']
```

inputs

name

property(*name*)

Decorator to make application properties.

Parameters **name** (*str*) – The name the property should take.

Examples

```
>>> class Foo(Brick):
...     @application
...     def apply(self, x):
...         return x + 1
...
...     @apply.property('inputs')
...     def apply_inputs(self):
...         return ['foo', 'bar']
>>> foo = Foo()
>>> foo.apply.inputs
['foo', 'bar']
```

class blocks.bricks.base.**ApplicationCall**(*application*)

Bases: blocks.graph.annotations.Annotation

A link between the variable tags and bricks.

The application call can be used to attach to an apply call auxiliary variables (e.g. monitors or regularizers) that do not form part of the main computation graph.

The application call object is created before the call to the application method and can be accessed by specifying an application_call argument.

Also see `Annotation`.

Parameters **`application`** (*`BoundApplication`* instance) – The bound application (i.e. belong to a brick instance) object being called

Examples

```
>>> class Foo(Brick):
...     @application
...     def apply(self, x, application_call):
...         application_call.add_auxiliary_variable(x.mean())
...         return x + 1
>>> x = tensor.vector()
>>> y = Foo().apply(x)
>>> from blocks.filter import get_application_call
>>> get_application_call(y)
<blocks.bricks.base.ApplicationCall object at ...>
```

`add_auxiliary_variable`(*variable*, *roles=None*, *name=None*)

Attach an auxiliary variable to the graph.

Auxiliary variables are Theano variables that are not part of a brick's output, but can be useful nonetheless e.g. as a regularizer or to monitor during training progress.

Parameters

- **`variable`** (`TensorVariable`) – The variable you want to add.
- **`roles`** (list of *`VariableRole`* instances, optional) – The roles of this variable. The `AUXILIARY` role will automatically be added. Other options are `COST`, `WEIGHT`, etc.
- **`name`** (*`str, optional`*) – Name to give to the variable. If the variable already has a name it will be overwritten.

Examples

```
>>> from blocks.bricks.base import application, Brick
>>> from blocks.roles import COST
>>> from blocks.utils import shared_floatx_nans
>>> class Foo(Brick):
...     def _allocate(self):
...         W = shared_floatx_nans((10, 10))
...         self.add_auxiliary_variable(W.mean(), name='mean_W')
...     @application
...     def apply(self, x, application_call):
...         application_call.add_auxiliary_variable(
...             x - 1, name='x_minus_1')
...         application_call.add_auxiliary_variable(
...             x.mean(), roles=[COST], name='mean_x')
```

(continues on next page)

(continued from previous page)

```
...         return x + 1
>>> from theano import tensor
>>> x = tensor.vector()
>>> y = Foo().apply(x)
>>> from blocks.graph import ComputationGraph
>>> cg = ComputationGraph([y])
>>> from blocks.filter import VariableFilter
>>> var_filter = VariableFilter(roles=[AUXILIARY])
>>> var_filter(cg.variables)
{x_minus_1, mean_W, mean_x}
>>> var_filter = VariableFilter(roles=[COST])
>>> var_filter(cg.variables)
{mean_x}
```

class blocks.bricks.base.**BoundApplication**(*application*, *brick*)

Bases: object

An application method bound to a *Brick* instance.

name

class blocks.bricks.base.**Brick**(*name=None*, *children=None*)

Bases: blocks.graph.annotations.Annotation

A brick encapsulates Theano operations with parameters.

A brick goes through the following stages:

1. Construction: The call to __init__() constructs a *Brick* instance with a name and creates any child bricks as well.
2. Allocation of parameters:
   (a) Allocation configuration of children: The *push_allocation_config()* method configures any children of this block.
   (b) Allocation: The *allocate()* method allocates the shared Theano variables required for the parameters. Also allocates parameters for all children.
3. The following can be done in either order:
   (a) Application: By applying the brick to a set of Theano variables a part of the computational graph of the final model is constructed.
   (b) The initialization of parameters:
      i. Initialization configuration of children: The *push_initialization_config()* method configures any children of this block.
      ii. Initialization: This sets the initial values of the parameters by a call to *initialize()*, which is needed to call the final compiled Theano function. Also initializes all children.

Not all stages need to be called explicitly. Step 3(a) will automatically allocate the parameters if needed. Similarly, step 3(b.2) and 2(b) will automatically perform steps 3(b.1) and 2(a) if needed. They only need to be called separately if greater control is required. The only two methods which always need to be called are an application method to construct the computational graph, and the *initialize()* method in order to initialize the parameters.

At each different stage, a brick might need a certain set of configuration settings. All of these settings can be passed to the __init__() constructor. However, by default many bricks support *lazy initialization*. This means that the configuration settings can be set later.

Note: Some arguments to `__init__()` are *always* required, even when lazy initialization is enabled. Other arguments must be given before calling *allocate()*, while others yet only need to be given in order to call *initialize()*. Always read the documentation of each brick carefully.

Lazy initialization can be turned off by setting `Brick.lazy = False`. In this case, there is no need to call *initialize()* manually anymore, but all the configuration must be passed to the `__init__()` method.

Parameters **name** (*str, optional*) – The name of this brick. This can be used to filter the application of certain modifications by brick names. By default, the brick receives the name of its class (lowercased).

name
str – The name of this brick.

print_shapes
bool – `False` by default. If `True` it logs the shapes of all the input and output variables, which can be useful for debugging.

parameters
list of `TensorSharedVariable` and `None` – After calling the *allocate()* method this attribute will be populated with the shared variables storing this brick's parameters. Allows for `None` so that parameters can always be accessed at the same index, even if some parameters are only defined given a particular configuration.

children
list of bricks – The children of this brick.

allocated
bool – `False` if *allocate()* has not been called yet. `True` otherwise.

initialized
bool – `False` if *allocate()* has not been called yet. `True` otherwise.

allocation_config_pushed
bool – `False` if *allocate()* or *push_allocation_config()* hasn't been called yet. `True` otherwise.

initialization_config_pushed
bool – `False` if *initialize()* or *push_initialization_config()* hasn't been called yet. `True` otherwise.

Notes

To provide support for lazy initialization, apply the *lazy()* decorator to the `__init__()` method.

Brick implementations *must* call the `__init__()` constructor of their parent using *super(BlockImplementation, self).__init__(**kwargs)* at the *beginning* of the overriding *__init__*.

The methods *_allocate()* and *_initialize()* need to be overridden if the brick needs to allocate shared variables and initialize their values in order to function.

A brick can have any number of methods which apply the brick on Theano variables. These methods should be decorated with the *application()* decorator.

If a brick has children, they must be listed in the *children* attribute. Moreover, if the brick wants to control the configuration of its children, the *_push_allocation_config()* and *_push_initialization_config()* methods need to be overridden.

Examples

Most bricks have lazy initialization enabled.

```
>>> import theano
>>> from blocks.initialization import IsotropicGaussian, Constant
>>> from blocks.bricks import Linear
>>> linear = Linear(input_dim=5, output_dim=3,
...                 weights_init=IsotropicGaussian(),
...                 biases_init=Constant(0))
>>> x = theano.tensor.vector()
>>> linear.apply(x)  # Calls linear.allocate() automatically
linear_apply_output
>>> linear.initialize()  # Initializes the weight matrix
```

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 34

_abc_registry = <_weakrefset.WeakSet object>

_allocate()
Brick implementation of parameter initialization.

Implement this if your brick needs to allocate its parameters.

> **Warning:** This method should never be called directly. Call *initialize()* instead.

_initialize()
Brick implementation of parameter initialization.

Implement this if your brick needs to initialize its parameters.

> **Warning:** This method should never be called directly. Call *initialize()* instead.

_push_allocation_config()
Brick implementation of configuring child before allocation.

Implement this if your brick needs to set the configuration of its children before allocation.

> **Warning:** This method should never be called directly. Call *push_allocation_config()* instead.

_push_initialization_config()
Brick implementation of configuring child before initialization.

Implement this if your brick needs to set the configuration of its children before initialization.

> **Warning:** This method should never be called directly. Call *push_initialization_config()* instead.

allocate()
Allocate shared variables for parameters.

Based on the current configuration of this *Brick* create Theano shared variables to store the parameters. After allocation, parameters are accessible through the *parameters* attribute.

This method calls the *allocate()* method of all children first, allowing the *_allocate()* method to override the parameters of the children if needed.

Raises ValueError – If the configuration of this brick is insufficient to determine the number of parameters or their dimensionality to be initialized.

Notes

This method sets the *parameters* attribute to an empty list. This is in order to ensure that calls to this method completely reset the parameters.

children

get_dim(*name*)
Get dimension of an input/output variable of a brick.

Parameters **name** (*str*) – The name of the variable.

get_dims(*names*)
Get list of dimensions for a set of input/output variables.

Parameters **names** (*list*) – The variable names.

Returns **dims** – The dimensions of the sources.

Return type list

get_hierarchical_name(*parameter*, *delimiter='/'*)
Return hierarhical name for a parameter.

Returns a path of the form `brick1/brick2/brick3.parameter1`. The delimiter is configurable.

Parameters **delimiter** (*str*) – The delimiter used to separate brick names in the path.

get_unique_path()
Returns unique path to this brick in the application graph.

initialize()
Initialize parameters.

Intialize parameters, such as weight matrices and biases.

Notes

If the brick has not allocated its parameters yet, this method will call the *allocate()* method in order to do so.

parameters

print_shapes = False
See *Brick.print_shapes*

push_allocation_config()
Push the configuration for allocation to child bricks.

Bricks can configure their children, based on their own current configuration. This will be automatically done by a call to *allocate()*, but if you want to override the configuration of child bricks manually, then you can call this function manually.

push_initialization_config()
Push the configuration for initialization to child bricks.

Bricks can configure their children, based on their own current configuration. This will be automatically done by a call to *initialize()*, but if you want to override the configuration of child bricks manually, then you can call this function manually.

class blocks.bricks.base.**Children**(*brick*, **args*, ***kwargs*)
Bases: *blocks.utils.containers.AnnotatingList*

Adds the brick to the list of parents of its children.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 34

_abc_registry = <_weakrefset.WeakSet object>

_delitem(*key*)
The operation to perform when an item is deleted.

_setitem(*key*, *value*)
The operation to perform when an item is inserted/appended.

class blocks.bricks.base.**LazyNone**(*name*)
Bases: object

class blocks.bricks.base.**Parameters**(*brick*, **args*, ***kwargs*)
Bases: *blocks.utils.containers.AnnotatingList*

Adds the PARAMETER role to parameters automatically.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 34

_abc_registry = <_weakrefset.WeakSet object>

_setitem(*key*, *value*)
The operation to perform when an item is inserted/appended.

class blocks.bricks.base.**_Brick**
Bases: abc.ABCMeta

Metaclass which attaches brick instances to the applications.

In addition picklability of *Application* objects is ensured. This means that *Application* objects can not be added to a brick class after it is created. To allow adding application methods programatically, the following hook is supported: the class namespace is searched for *decorators* attribute, which can contain a list of functions to be applied to the namespace of the class being created. These functions can arbitratily modify this namespace.

blocks.bricks.base.**_variable_name**(*brick_name*, *application_name*, *name*)

blocks.bricks.base.**application**(**args*, ***kwargs*)
Decorator for methods that apply a brick to inputs.

Parameters

- **optional** (***kwargs,*) – The application method to wrap.
- **optional** – Attributes to attach to this application.

Notes

This decorator replaces application methods with *Application* instances. It also sets the attributes given as keyword arguments to the decorator.

Note that this decorator purposely does not wrap the original method using e.g. `wraps()` or `update_wrapper()`, since that would make the class impossible to pickle (see notes at *Application*).

Examples

```
>>> class Foo(Brick):
...     @application(inputs=['x'], outputs=['y'])
...     def apply(self, x):
...         return x + 1
...     @application
...     def other_apply(self, x):
...         return x - 1
>>> foo = Foo()
>>> Foo.apply.inputs
['x']
>>> foo.apply.outputs
['y']
>>> Foo.other_apply
<blocks.bricks.base.Application object at ...>
```

blocks.bricks.base.**args_to_kwargs**(*args*, *f*)

blocks.bricks.base.**copy_and_tag**(*variable*, *brick*, *call*, *role*, *application_name*, *name*)
Helper method to copy a variable and annotate it.

blocks.bricks.base.**create_unbound_method**(*func*, *cls*)
Create an unbounded method from a function and a class.

Notes

See https://bitbucket.org/gutworth/six/pull-request/64.

blocks.bricks.base.**lazy**(*allocation=None*, *initialization=None*)
Makes the initialization lazy.

This decorator allows the user to define positional arguments which will not be needed until the allocation or initialization stage of the brick. If these arguments are not passed, it will automatically replace them with a custom `None` object. It is assumed that the missing arguments can be set after initialization by setting attributes with the same name.

Parameters

- **allocation** (*list*) – A list of argument names that are needed for allocation.
- **initialization** (*list*) – A list of argument names that are needed for initialization.

Examples

```
>>> class SomeBrick(Brick):
...     @lazy(allocation=['a'], initialization=['b'])
...     def __init__(self, a, b, c='c', d=None):
...         print(a, b, c, d)
>>> brick = SomeBrick('a')
a NoneInitialization c None
>>> brick = SomeBrick(d='d', b='b')
NoneAllocation b c d
```

blocks.bricks.base.**rename_function**(*function*, *new_name*)

class blocks.bricks.**Activation**(*name=None*, *children=None*)
Bases: *blocks.bricks.base.Brick*

Elementwise application of activation function.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 34

_abc_registry = <_weakrefset.WeakSet object>

class blocks.bricks.interfaces.**ActivationDocumentation**
Bases: *blocks.bricks.base._Brick*

Dynamically adds documentation to activations.

Notes

See http://bugs.python.org/issue12773.

Extensions

class blocks.extensions.predicates.**OnLogRecord**(*record_name*)
Bases: object

Trigger a callback when a certain log record is found.

Parameters **record_name** (*str*) – The record name to check.

class blocks.monitoring.evaluators.**AggregationBuffer**(*variables*, *use_take_last=False*)
Bases: object

Intermediate results of aggregating values of Theano variables.

Encapsulates aggregators for a list of Theano variables. Collects the respective updates and provides initialization and readout routines.

Parameters

- **variables** (list of TensorVariable) – The variable names are used as record names in the logs. Hence, all the variable names must be unique.
- **use_take_last** (*bool*) – When True, the TakeLast aggregation scheme is used instead of _DataIndependent for those variables that do not require data to be computed.

initialization_updates
list of tuples – Initialization updates of the aggregators.

accumulation_updates
list of tuples – Accumulation updates of the aggregators.

readout_variables
dict – A dictionary of record names to `TensorVariable` representing the aggregated values.

inputs
list of `TensorVariable` – The list of inputs needed for accumulation.

_compile()
Compiles Theano functions.

Todo: The current compilation method does not account for updates attached to *ComputationGraph* elements. Compiling should be out-sourced to *ComputationGraph* to deal with it.

_create_aggregators()
Create aggregators and collect updates.

get_aggregated_values()
Readout the aggregated values.

initialize_aggregators()
Initialize the aggregators.

class blocks.monitoring.evaluators.**DatasetEvaluator**(*variables*, *updates=None*)
Bases: `object`

A DatasetEvaluator evaluates many Theano variables or other quantities.

The DatasetEvaluator provides a do-it-all method, *evaluate()*, which computes values of `variables` on a dataset.

Alternatively, methods *initialize_aggregators()*, *process_batch()*, *get_aggregated_values()* can be used with a custom loop over data.

The values computed on subsets of the given dataset are aggregated using the `AggregationScheme`s provided in the `aggregation_scheme` tags. If no tag is given, the value is **averaged over minibatches**. However, care is taken to ensure that variables which do not depend on data are not unnecessarily recomputed.

Parameters

- **variables** (list of `TensorVariable` and) – `MonitoredQuantity` The variable names are used as record names in the logs. Hence, all the names must be unique.

  Each variable can be tagged with an `AggregationScheme` that specifies how the value can be computed for a data set by aggregating minibatches.

- **updates** (list of tuples or `OrderedDict` or None) – `TensorSharedVariable` updates to be performed during evaluation. This parameter is only for Theano variables. Be careful not to update any model parameters as this is not intended to alter your model in any meaningfullway. A typical use case of this option arises when the theano function used for evaluation contains a call to:function:*~theano.scan* which might have returned shared variable updates.

_compile()
Compiles Theano functions.

Todo: The current compilation method does not account for updates attached to *ComputationGraph* elements. Compiling should be out-sourced to *ComputationGraph* to deal with it.

evaluate(*data_stream*)
Compute the variables over a data stream.

Parameters **data_stream** (instance of DataStream) – The data stream. Only the first epoch of data is used.

Returns

- *A mapping from record names to the values computed on the provided*
- *dataset.*

get_aggregated_values()

initialize_aggregators()

process_batch(*batch*)

class blocks.monitoring.evaluators.**MonitoredQuantityBuffer**(*quantities*)
Bases: object

Intermediate results of aggregating values of monitored-quantity.

Aggregate results for a list of monitored-quantity for every single batch. Provides initialization and readout routines to initialize each quantity and capture its aggregated results.

Parameters **quantities** (list of MonitoredQuantity) – The quantity names are used as record names in the logs. Hence, all the quantity names must be unique.

requires
list of TensorVariable – Needed to calculate monitored-quantities.

quantity_names
list of str – Names of quantities.

inputs
list of TensorVariable – The list of inputs needed for variables in *requires*.

aggregate_quantities(*numerical_values*)
Aggregate the results for every batch.

get_aggregated_values()
Get the aggregated values.

initialize_quantities()
Initialize the quantities.

blocks.monitoring.evaluators.**_validate_variable_names**(*variables*)
Check for missing and duplicate variable names.

Utils

class blocks.utils.containers.**AnnotatingList**(*items=None*)
Bases: _abcoll.MutableSequence

Mutable sequence performing operations on inserted/removed items.

Parameters **items** (*iterable, optional*) – An iterable of items to initialize the sequence with.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 34

_abc_registry = <_weakrefset.WeakSet object>

_delitem(*key*)
The operation to perform when an item is deleted.

_setitem(*key*, *value*)
The operation to perform when an item is inserted/appended.

insert(*key*, *value*)
S.insert(index, object) – insert object before index

class blocks.utils.profile.**Profile**
Bases: object

A profile of hierarchical timers.

Keeps track of timings performed with *Timer*. It also keeps track of the way these timings were nested and makes use of this information when reporting.

enter(*name*)

exit(*t*)

report(*f=<open file '<stderr>', mode 'w'>*)
Print a report of timing information to standard output.

Parameters **f** (*object, optional*) – An object with a `write` method that accepts string inputs. Can be a file object, `sys.stdout`, etc. Defaults to `sys.stderr`.

class blocks.utils.profile.**Timer**(*name*, *profile*)
Bases: object

A context manager to time the execution time of code within it.

This timer is attached to a *Profile* object that it reports timings to. The *Profile* object accumulates the timings. Timers can be nested, which the *Profile* will automatically keep track of and use in its reporting.

Parameters

- **name** (*str*) – The name of this section. Expected to adhere to variable naming styles.
- **profile** (*Profile*) – The profile of the main loop. This is the object this context manager will report the execution time to. The accumulation and processing of timing information is handled by this object.

Notes

Timings are reported using `timeit.default_timer()`.

Building documentation

If you've made significant changes to the documentation, you can build a local to see how your changes are rendered. You will need to install Sphinx, the Napoleon extension (to enable NumPy docstring support), and the Read the Docs theme. You can do this by installing the optional `docs` requirements.

For Blocks:

```
$ pip install --upgrade git+git://github.com/user/blocks.git#egg=blocks[docs]
```

For Fuel:

```
$ pip install --upgrade git+git://github.com/user/fuel.git#egg=fuel[docs]
```

After the requirements have been installed, you can build a copy of the documentation by running the following command from the root `blocks` (or `fuel`) directory.

```
$ sphinx-build -b html docs docs/_build/html
```

Docstrings

Blocks and Fuel follow the NumPy docstring standards. For a quick introduction, have a look at the NumPy or Napoleon examples of compliant docstrings. A few common mistakes to avoid:

- There is no line break after the opening quotes (`"""`).
- There is an empty line before the closing quotes (`"""`).
- The summary should not be more than one line.

The docstrings are formatted using reStructuredText, and can make use of all the formatting capabilities this provides. They are rendered into HTML documentation using the Read the Docs service. After code has been merged, please ensure that documentation was built successfully and that your docstrings rendered as you intended by looking at the online documentation (for Blocks or Fuel, which is automatically updated.

Writing doctests is encouraged, and they are run as part of the test suite. They should use Python 3 syntax.

References and Intersphinx

Sphinx allows you to reference other objects in the framework. This automatically creates links to the API documentation of that object (if it exists).

```
This is a link to :class:`SomeClass` in the same file. If you want to
reference an object in another file, you can use a leading dot to tell
Sphinx to look in all files e.g. :meth:`.SomeClass.a_method`.
```

Intersphinx is an extension that is enabled which allows to you to reference the documentation of other projects such as Theano, NumPy and Scipy.

```
The input to a method can be of the type :class:`~numpy.ndarray`. Note that
in this case we need to give the full path. The tilde (~) tells Sphinx not
to render the full path (numpy.ndarray), but only the object itself
(ndarray).
```

Warning: Because of a bug in Napoleon you can't use the reference to a type in the "Returns" section of your docstring without giving it a name. This doesn't render correctly:

```
Returns
-------
:class:`Brick`
    The returned Brick.
```

But this does:

```
Returns
-------
retured_brick : :class:`Brick`
    The returned Brick.
```

Pull request workflow

Blocks development takes place on GitHub; developers (including project leads!) add new features by sending pull requests from their personal fork (we operate on the so-called fork & pull model).

This page serves as a "quick reference" for the recommended pull request workflow. It assumes you are working on a UNIX-like environment with Git already installed. It is **not** intended to be an exhaustive tutorial on Git; there are many of those available.

Before you begin

Create a GitHub account

If you don't already have one, you should create yourself a GitHub account.

Fork the Blocks repository

Once you've set up your account and logged in, you should fork the Blocks repository to your account by clicking the "Fork" button on the official repository's web page. More information on forking is available in the GitHub documentation.

Clone from your fork

In the side bar of your newly created fork of the Blocks repository, you should see a field that says **HTTPS clone URL** above it. Copy that to your clipboard and run, at the terminal,

```
$ git clone CLONE_URL
```

where `CLONE_URL` is the URL you copied from your GitHub fork.

If you're doing a lot of development with GitHub you should look into setting up SSH key authentication.

Add the official Blocks repository as a remote

In order to keep up with changes to the official Blocks repository, notify Git of its existence and location by running

```
$ git remote add upstream https://github.com/mila-udem/blocks.git
```

You only need to do this once.

Beginning a pull request

Verify that origin points to your fork

Running the command

```
$ git remote -v | grep origin
```

should display two lines. The URLs therein should contain your GitHub username.

Update your upstream remote

Your cloned repository stores a local history of the activity in remote repositories, and only interacts with the Internet when certain commands are invoked. In order to synchronize the activity in the official Blocks repository (which Git now knows as `upstream`) with the local mirror of the history related to `upstream`, run

```
$ git fetch upstream
```

You should do this before starting every pull request, for reasons that will become clear below.

Create a new branch for your pull request based on the latest development version of Blocks

In order to create a new branch *starting from the latest commit in the master branch of the official Blocks repository*, make sure you've fetched from `upstream` (see above) and run

```
$ git checkout -b my_branch_name_for_my_cool_feature upstream/master
```

Obviously, you'll probably want to choose a better branch name.

Note that doing this (rather than simply creating a new branch from some arbtirary point) may save you from a (possibly painful) rebase later on.

Working on your pull request

Make modifications, stage them, and commit them

Repeat until satisfied:

- Make some modifications to the code
- Stage them using `git add` (`git add -p` is particularly useful)
- `git commit` them, alternately `git reset` to undo staging by `git add`.

Push the branch to your fork

```
$ git push -u origin my_branch_name_for_my_cool_feature
```

Submitting for review

Send a pull request

This can be done from the GitHub web interface for your fork. See this documentation from GitHub for more information.

Give your pull request an appropriate title which makes it obvious what the content is. **If it is intended to resolve a specific ticket**, put "Fixes #NNN." in the pull request description field, where *NNN* is the issue number. By doing this, GitHub will know to automatically close the issue when your pull request is merged.

Blocks development occurs in two separate branches: The `master` branch is the development branch. If you want to contribute a new feature or change the behavior of Blocks in any way, please make your pull request to this branch.

The `stable` branch contains the latest release of Blocks. If you are fixing a bug (that is present in the latest release), make a pull request to this branch. If the bug is present in both the `master` and `stable` branch, two separate pull requests are in order. The command `git-cherry-pick_` could be useful here.

Incorporating feedback

In order to add additional commits responding to reviewer feedback, simply follow the instructions above for using `git add` and `git commit`, and finally `git push` (after running the initial command with `-u`, you should simply be able to use `git push` without any further arguments).

Rebasing

Occasionally you will be asked to *rebase* your branch against the latest master. To do this, run (while you have your branch checked out)

```
$ git fetch upstream && git rebase upstream/master
```

You may encounter an error message about one or more *conflicts*. See GitHub's help page on the subject. Note that after a rebase you will usually have to overwrite previous commits on your fork's copy of the branch with `git push --force`.

CHAPTER 3

Quickstart

Construct your model.

```
>>> mlp = MLP(activations=[Tanh(), Softmax()], dims=[784, 100, 10],
...           weights_init=IsotropicGaussian(0.01), biases_init=Constant(0))
>>> mlp.initialize()
```

Calculate your loss function.

```
>>> x = tensor.matrix('features')
>>> y = tensor.lmatrix('targets')
>>> y_hat = mlp.apply(x)
>>> cost = CategoricalCrossEntropy().apply(y.flatten(), y_hat)
>>> error_rate = MisclassificationRate().apply(y.flatten(), y_hat)
```

Load your training data using Fuel.

```
>>> mnist_train = MNIST(("train",))
>>> train_stream = Flatten(
...     DataStream.default_stream(
...         dataset=mnist_train,
...         iteration_scheme=SequentialScheme(mnist_train.num_examples, 128)),
...     which_sources=('features',))
>>> mnist_test = MNIST(("test",))
>>> test_stream = Flatten(
...     DataStream.default_stream(
...         dataset=mnist_test,
...         iteration_scheme=SequentialScheme(mnist_test.num_examples, 1024)),
...     which_sources=('features',))
```

And train!

```
>>> from blocks.model import Model
>>> main_loop = MainLoop(
...     model=Model(cost), data_stream=train_stream,
```

(continues on next page)

(continued from previous page)

```
...     algorithm=GradientDescent(
...         cost=cost, parameters=ComputationGraph(cost).parameters,
...         step_rule=Scale(learning_rate=0.1)),
...     extensions=[FinishAfter(after_n_epochs=5),
...                 DataStreamMonitoring(
...                     variables=[cost, error_rate],
...                     data_stream=test_stream,
...                     prefix="test"),
...                 Printing()])
>>> main_loop.run()

...
```

For a runnable version of this code, please see the MNIST demo in our repository with examples.

3.1 Features

Currently Blocks supports and provides:

- Constructing parametrized Theano operations, called "bricks"
- Pattern matching to select variables and bricks in large models
- Algorithms to optimize your model
- Saving and resuming of training
- Monitoring and analyzing values during training progress (on the training set as well as on test sets)
- Application of graph transformations, such as dropout (*limited support*)

In the future we also hope to support:

- Dimension, type and axes-checking

CHAPTER 4

Indices and tables

- genindex
- modindex

Bibliography

[ADADELTA] Matthew D. Zeiler, *ADADELTA: An Adaptive Learning Rate Method*, arXiv:1212.5701.

[ADAGRAD] Duchi J, Hazan E, Singer Y., *Adaptive subgradient methods for online learning and stochastic optimization*, http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[King2014] Diederik Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, http://arxiv.org/abs/1412.6980

[Hint2014] Geoff Hinton, *Neural Networks for Machine Learning*, lecture 6a, http://cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[Srebro2005] Nathan Srebro and Adi Shraibman. "Rank, Trace-Norm and Max-Norm". *18th Annual Conference on Learning Theory (COLT)*, June 2005.

[Hinton2012] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, Ruslan R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors". arXiv:1207.0580.

[BN] Sergey Ioffe and Christian Szegedy. *Batch normalization: accelerating deep network training by reducing internal covariate shift*. ICML (2015), pp. 448-456.

[SK2016] Tim Salimans and Diederik P. Kingma. *Weight normalization: a simple reparameterization to accelerate training of deep neural networks*. arXiv 1602.07868.

[GWFM13] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio, *Maxout networks*, ICML (2013), pp. 1319-1327.

[cuDNN] NVIDIA cuDNN.

[CvMG14] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, EMNLP (2014), pp. 1724-1734.

[GSS03] Gers, Felix A., Nicol N. Schraudolph, and Jürgen Schmidhuber, *Learning precise timing with LSTM recurrent networks*, Journal of Machine Learning Research 3 (2003), pp. 115-143.

[Grav13] Graves, Alex, *Generating sequences with recurrent neural networks*, arXiv preprint arXiv:1308.0850 (2013).

[HS97] Sepp Hochreiter, and Jürgen Schmidhuber, *Long Short-Term Memory*, Neural Computation 9(8) (1997), pp. 1735-1780.

[BCB] Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate.

[DROPOUT] Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*, arXiv:1207.0580.

[Saxe2013] Saxe, A.M., McClelland, J.L., Ganguli, S., 2013., *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*, arXiv:1312.6120 [cond-mat, q-bio, stat].

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W