Blocks Documentation

Release 0.0.1

Université de Montréal

October 08, 2015

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 The last stable (and thus likely an outdated) version can be found in the `stable` branch.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 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/req.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/req.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 `req.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Warning: Pip may try to install or update NumPy and SciPy if they are not present or outdated.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_`).

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/req.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 req.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. 1.1: 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.bricks 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 `Brick.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 `Brick.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
>>> 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 `Brick.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 `Brick.initialize()`, it automatically pushed the weight matrix and biases initialization configuration to its children.

```
>>> mlp.initialize()
>>> mlp.children[1].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 `Brick.push_allocation_config()` and `Brick.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 $[0, 1]$.

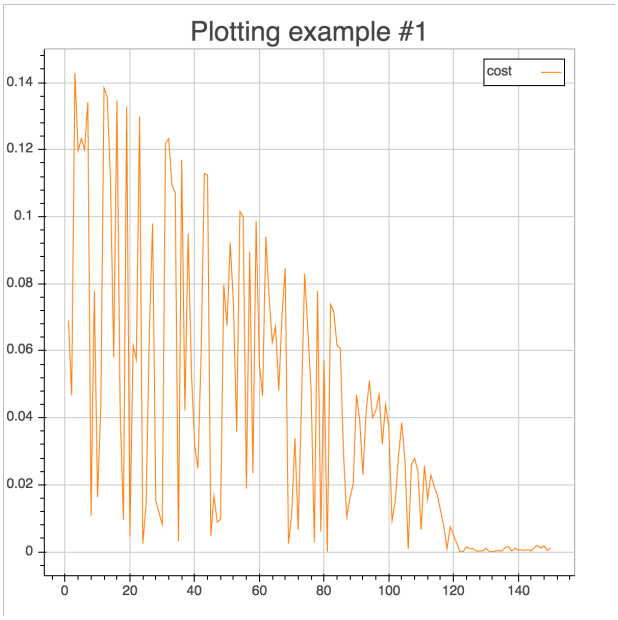
```
>>> import numpy
>>> from fuel.streams import DataStream
>>> from fuel.datasets import IterableDataset
>>> data_stream = DataStream(IterableDataset(
...     numpy.random.rand(150).astype(theano.config.floatX)))
```

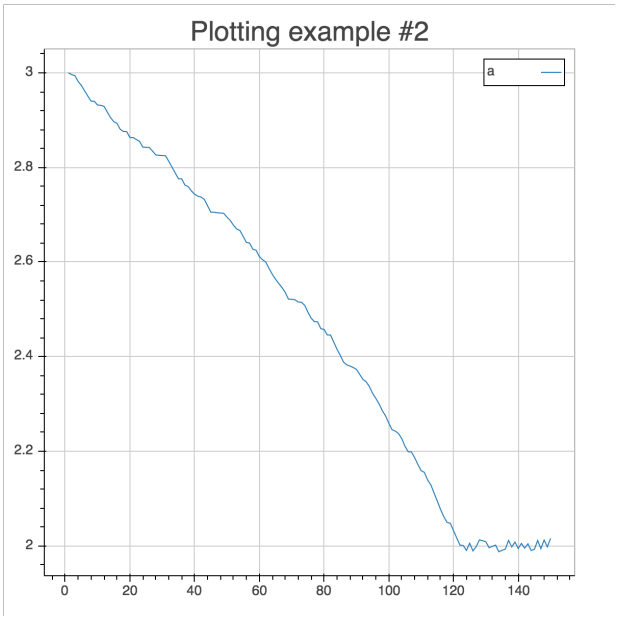
Now let's train with gradient descent and plot the results.

```
>>> 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

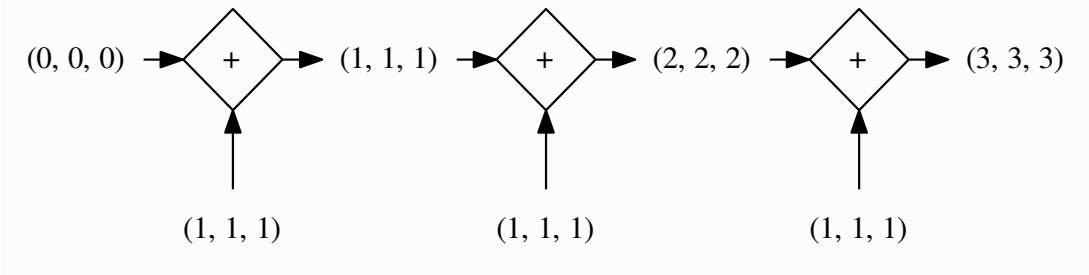
In-depth

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



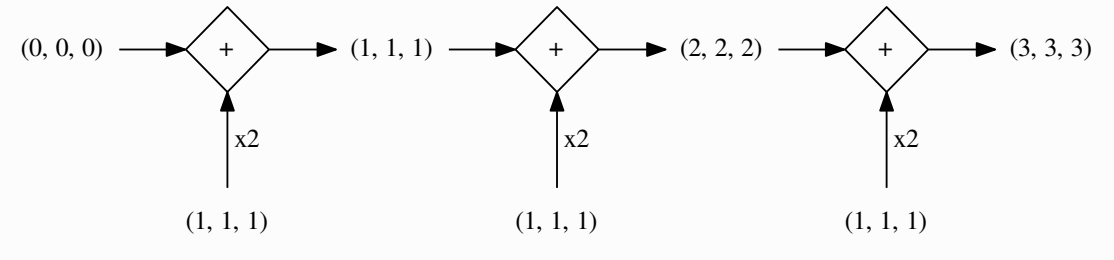
As a starting example, we'll be building an RNN which accumulates the input it receives (figure above). The equation describing that RNN is

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

```
>>> 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

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

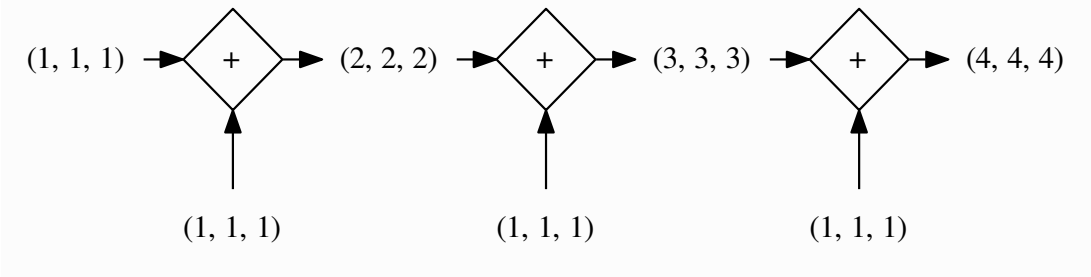
```
>>> from blocks.bricks import Linear
>>> doubler = Linear(
...     input_dim=3, output_dim=3, weights_init=initialization.Identity(2),
...     biases_init=initialization.Constant(0))
>>> doubler.initialize()
>>> h_doubler = rnn.apply(doubler.apply(x))
>>> f = theano.function([x], h_doubler)
>>> print(f(numpy.ones((3,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

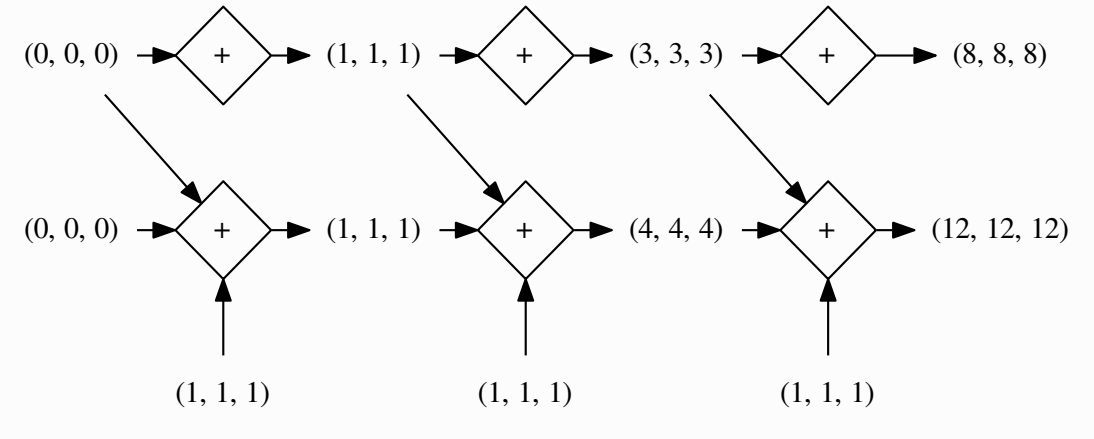
Say something about the `return_initial_states` argument

2.1.5 Iterate (or not)

The `apply` method of a recurrent brick accepts an `iterate` argument, which defaults to `True`. Setting it 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
...
...     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`.

2.2 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
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 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.

2.3.1 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.

2.3.2 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 arcive 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.

2.4 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.

2.4.1 Algorithms

2.4.2 Bricks

Convolutional bricks

Routing bricks

Recurrent bricks

Attention bricks

Sequence generators

Cost bricks

Wrapper bricks

2.4.3 Extensions

class blocks.extensions.**CallbackName**
Bases: str

A name of a TrainingExtension callback.

Raises

- class:*TypeError* on comparison with a string which is not a name of
- **TrainingExtension callback.**

class blocks.extensions.**FinishAfter**(**kwargs)
Bases: *blocks.extensions.SimpleExtension*

Finishes the training process when triggered.

do(*which_callback*, **args*)

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*)

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()

before_batch(*batch*)

before_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(['after_batch', 'after_training', 'before_epoch', 'before_training', 'before_first_epoch'

INTEGER_TRIGGERS = frozenset(['every_n_batches', 'after_n_epochs', 'every_n_epochs', 'after_n_batches'])

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.

Parameters

- **the** (*See*) –
- **parameters.** (*possible*) –

class blocks.extensions.**Timing**(***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.

Notes

Add this extension *before* the *Printing* extension.

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*)

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()
The callback invoked when an error occurs.

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

Training

class blocks.extensions.training.**SharedVariableModifier**(*parameter*, *function*, ***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*).

do(*which_callback*, **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*)

Serialization

2.4.4 Filter

2.4.5 Computational graph

2.4.6 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*)

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*)

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*)

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].
- **.** – [Saxe2014] 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].

generate(*rng*, *shape*)

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*)

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*)

2.4.7 Logging

2.4.8 Main loop

2.4.9 Model

2.4.10 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

2.4.11 Brick selectors

2.4.12 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*)

Computes the total L2 norm of a set of tensors.

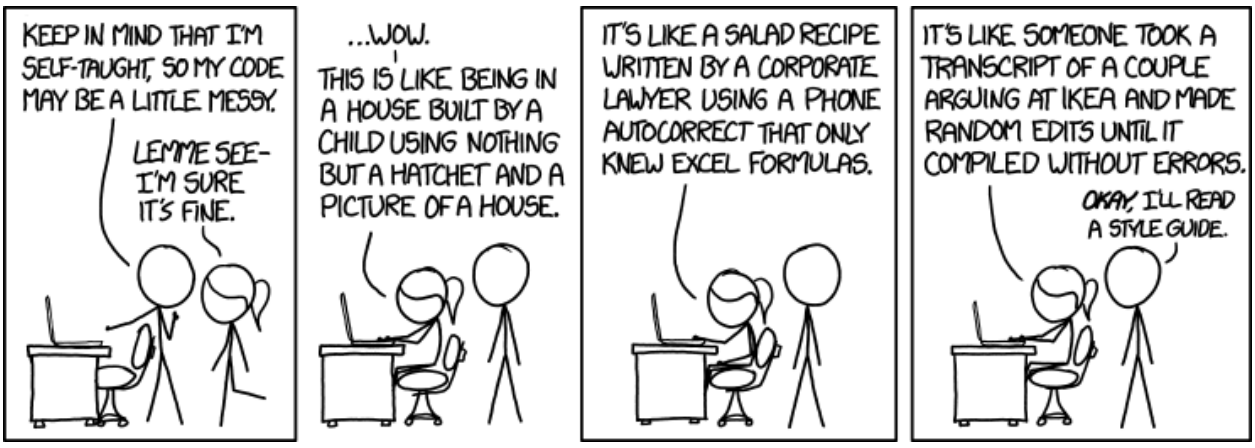
Converts all operands to `TensorVariable` (see `as_tensor_variable()`).

Parameters **`tensors`** (iterable of `TensorVariable` (or compatible)) – The tensors.

2.4.13 Utilities

2.5 Development

We want to encourage everyone to contribute to the development of Blocks. 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!



2.5.1 Formatting guidelines

Blocks follows the PEP8 style guide closely, so please make sure you are familiar with it. Our Travis CI buildbot runs 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:

- 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
```

2.5.2 Code guidelines

Some guidelines to keep in mind when coding for Blocks. 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 aims 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 tries 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
```

2.5.3 Unit testing

Blocks uses 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.

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].

2.5.4 Writing and building documentation

The documentation guidelines outline how to write documentation for Blocks, and how to build a local copy of the documentation for testing purposes.

2.5.5 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.

[1] For all tests but the doctests, nose can also be used.

2.5.6 Installation

See the instructions at the bottom of the installation instructions.

2.5.7 Sending a pull request

See our pull request workflow for a refresher on the general recipe for sending a pull request to Blocks.

Internal API

- *Bricks*
- *Extensions*
- *Utils*

Bricks

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.

Utils

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:

```
$ pip install --upgrade git+git://github.com/user/blocks.git#egg=blocks[docs]
```

After the requirements have been installed, you can build a copy of the documentation by running the following command from the root `blocks` directory.

```
$ sphinx-build -b html docs docs/_build/html
```

Docstrings

Blocks follows 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, 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,
...     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

b

A

B

C

D

F

G