lasagne Documentation

Release 0.1

Lasagne contributors

August 13, 2015

Contents

Lasagne is a lightweight library to build and train neural networks in Theano.

Lasagne is a work in progress, input is welcome. The available documentation is limited for now. The project is on GitHub.

CHAPTER 1

User Guide

The Lasagne user guide explains how to install Lasagne, how to build and train neural networks using Lasagne, and how to contribute to the library as a developer.

1.1 Installation

Lasagne has a couple of prerequisites that need to be installed first, but it is not very picky about versions. The single exception is Theano: Due to its tight coupling to Theano, you will have to install a recent version of Theano (usually more recent than the latest official release!) fitting the version of Lasagne you choose to install.

Most of the instructions below assume you are running a Linux or Mac system; please do not hesitate to suggest instructions for Windows via the *Edit on GitHub* link on the top right!

If you run into any trouble, please check the Theano installation instructions which cover installing the prerequisites for a range of operating systems, or ask for help on our mailing list.

1.1.1 Prerequisites

Python + pip

Lasagne currently requires Python 2.7 or 3.4 to run. Please install Python via the package manager of your operating system if it is not included already.

Python includes `pip` for installing additional modules that are not shipped with your operating system, or shipped in an old version, and we will make use of it below. We recommend installing these modules into your home directory via `--user`, or into a virtual environment via `virtualenv`.

C compiler

Theano requires a working C compiler, and numpy/scipy require a compiler as well if you install them via `pip`. On Linux, the default compiler is usually `gcc`, and on Mac OS, it's `clang`. Again, please install them via the package manager of your operating system.

numpy/scipy + BLAS

Lasagne requires numpy of version 1.6.2 or above, and Theano also requires scipy 0.11 or above. Numpy/scipy rely on a BLAS library to provide fast linear algebra routines. They will work fine without one, but a lot slower, so it is worth getting this right (but this is less important if you plan to use a GPU).

If you install numpy and scipy via your operating system's package manager, they should link to the BLAS library installed in your system. If you install numpy and scipy via `pip install numpy` and `pip install scipy`, make sure to have development headers for your BLAS library installed (e.g., the `libopenblas-dev` package on Debian/Ubuntu) while running the installation command. Please refer to the numpy/scipy build instructions if in doubt.

Theano

The version to install depends on the Lasagne version you choose, so this will be handled below.

1.1.2 Stable Lasagne release

Lasagne 0.1 requires a more recent version of Theano than the one available on PyPI. To install a version that is known to work, run the following command:

```
pip install -r https://raw.githubusercontent.com/Lasagne/Lasagne/v0.1/requirements.txt
```

> **Warning:** An even more recent version of Theano will often work as well, but at the time of writing, a simple `pip install Theano` will give you a version that is too old.

To install release 0.1 of Lasagne from PyPI, run the following command:

```
pip install Lasagne==0.1
```

If you do not use `virtualenv`, add `--user` to both commands to install into your home directory instead. To upgrade from an earlier installation, add `--upgrade`.

1.1.3 Bleeding-edge version

The latest development version of Lasagne usually works fine with the latest development version of Theano. To install both, run the following commands:

```
pip install --upgrade https://github.com/Theano/Theano/archive/master.zip
pip install --upgrade https://github.com/Lasagne/Lasagne/archive/master.zip
```

Again, add `--user` if you want to install to your home directory instead.

1.1.4 Development installation

Alternatively, you can install Lasagne (and optionally Theano) from source, in a way that any changes to your local copy of the source tree take effect without requiring a reinstall. This is often referred to as *editable* or *development* mode. Firstly, you will need to obtain a copy of the source tree:

```
git clone https://github.com/Lasagne/Lasagne.git
```

It will be cloned to a subdirectory called `Lasagne`. Make sure to place it in some permanent location, as for an *editable* installation, Python will import the module directly from this directory and not copy over the files. Enter the directory and install the known good version of Theano:

```
cd Lasagne
pip install -r requirements.txt
```

Alternatively, install the bleeding-edge version of Theano as described in the previous section.

To install the Lasagne package itself, in editable mode, run:

```
pip install --editable .
```

As always, add `--user` to install it to your home directory instead.

Optional: If you plan to contribute to Lasagne, you will need to fork the Lasagne repository on GitHub. This will create a repository under your user account. Update your local clone to refer to the official repository as `upstream`, and your personal fork as `origin`:

```
git remote rename origin upstream
git remote add origin https://github.com/<your-github-name>/Lasagne.git
```

If you set up an SSH key, use the SSH clone URL instead: `git@github.com:<your-github-name>/Lasagne.git`.

You can now use this installation to develop features and send us pull requests on GitHub, see *Development*!

1.1.5 GPU support

Thanks to Theano, Lasagne transparently supports training your networks on a GPU, which may be 10 to 50 times faster than training them on a CPU. Currently, this requires an NVIDIA GPU with CUDA support, and some additional software for Theano to use it.

CUDA

Install the latest CUDA Toolkit and possibly the corresponding driver available from NVIDIA: https://developer.nvidia.com/cuda-downloads

Closely follow the *Getting Started Guide* linked underneath the download table to be sure you don't mess up your system by installing conflicting drivers.

After installation, make sure `/usr/local/cuda/bin` is in your `PATH`, so `nvcc --version` works. Also make sure `/usr/local/cuda/lib64` is in your `LD_LIBRARY_PATH`, so the toolkit libraries can be found.

Theano

If CUDA is set up correctly, the following should print some information on your GPU (the first CUDA-capable GPU in your system if you have multiple ones):

```
THEANO_FLAGS=device=gpu python -c "import theano; print theano.sandbox.cuda.device_properties(0)"
```

To configure Theano to use the GPU by default, create a file `.theanorc` directly in your home directory, with the following contents:

```
[global]
floatX = float32
device = gpu
```

Optionally add `allow_gc = False` for some extra performance at the expense of (sometimes substantially) higher GPU memory usage.

If you run into problems, please check Theano's instructions for Using the GPU.

cuDNN

NVIDIA provides a library for common neural network operations that especially speeds up Convolutional Neural Networks (CNNs). Again, it can be obtained from NVIDIA (after registering as a developer): https://developer.nvidia.com/cudnn

Note that it requires a reasonably modern GPU with Compute Capability 3.0 or higher; see NVIDIA's list of CUDA GPUs.

To install it, copy the `*.h` files to `/usr/local/cuda/include` and the `lib*` files to `/usr/local/cuda/lib64`.

To check whether it is found by Theano, run the following command:

```
python -c "import theano; print theano.sandbox.cuda.dnn.dnn_available() or theano.sandbox.cuda.dnn.d
```

It will print `True` if everything is fine, or an error message otherwise. There are no additional steps required for Theano to make use of cuDNN.

1.2 Tutorial

This tutorial will walk you through building a handwritten digits classifier using the MNIST dataset, arguably the "Hello World" of neural networks. More tutorials and examples can be found in the Lasagne Recipes repository.

1.2.1 Before we start

The tutorial assumes that you are somewhat familiar with neural networks and Theano (the library which Lasagne is built on top of). You can try to learn both at once from the Deeplearning Tutorial.

For a more slow-paced introduction to artificial neural networks, we recommend Convolutional Neural Networks for Visual Recognition by Andrej Karpathy et al., Neural Networks and Deep Learning by Michael Nielsen or a standard text book such as "Machine Learning" by Tom Mitchell.

To learn more about Theano, have a look at the Theano tutorial. You will not need all of it, but a basic understanding of how Theano works is required to be able to use Lasagne. If you're new to Theano, going through that tutorial up to (and including) "Graph Structures" should get you covered!

1.2.2 Run the MNIST example

In this first part of the tutorial, we will just run the MNIST example that's included in the source distribution of Lasagne.

We assume that you have already run through the *Installation*. If you haven't done so already, get a copy of the source tree of Lasagne, and navigate to the folder in a terminal window. Enter the `examples` folder and run the `mnist.py` example script:

```
cd examples
python mnist.py
```

If everything is set up correctly, you will get an output like the following:

```
Using gpu device 0: GeForce GT 640
Loading data...
Downloading MNIST dataset...
Building model and compiling functions...
```

```
Starting training...

Epoch 1 of 500 took 1.858s
  training loss:                1.233348
  validation loss:              0.405868
  validation accuracy:          88.78 %
Epoch 2 of 500 took 1.845s
  training loss:                0.571644
  validation loss:              0.310221
  validation accuracy:          91.24 %
Epoch 3 of 500 took 1.845s
  training loss:                0.471582
  validation loss:              0.265931
  validation accuracy:          92.35 %
Epoch 4 of 500 took 1.847s
  training loss:                0.412204
  validation loss:              0.238558
  validation accuracy:          93.05 %
...
```

The example script allows you to try three different models, selected via the first command line argument. Run the script with `python mnist.py --help` for more information and feel free to play around with it some more before we have a look at the implementation.

1.2.3 Understand the MNIST example

Let's now investigate what's needed to make that happen! To follow along, open up the source code in your favorite editor (or online: mnist.py).

Preface

The first thing you might notice is that besides Lasagne, we also import numpy and Theano:

```
import numpy as np
import theano
import theano.tensor as T

import lasagne
```

While Lasagne is built on top of Theano, it is meant as a supplement helping with some tasks, not as a replacement. You will always mix Lasagne with some vanilla Theano code.

Loading data

The first piece of code defines a function `load_dataset()`. Its purpose is to download the MNIST dataset (if it hasn't been downloaded yet) and return it in the form of regular numpy arrays. There is no Lasagne involved at all, so for the purpose of this tutorial, we can regard it as:

```
def load_dataset():
    ...
    return X_train, y_train, X_val, y_val, X_test, y_test
```

`X_train.shape` is `(50000, 1, 28, 28)`, to be interpreted as: 50,000 images of 1 channel, 28 rows and 28 columns each. Note that the number of channels is 1 because we have monochrome input. Color images would have 3 channels, spectrograms also would have a single channel. `y_train.shape` is simply `(50000,)`, that is, it is a

vector the same length of `X_train` giving an integer class label for each image – namely, the digit between 0 and 9 depicted in the image (according to the human annotator who drew that digit).

Building the model

This is where Lasagne steps in. It allows you to define an arbitrarily structured neural network by creating and stacking or merging layers. Since every layer knows its immediate incoming layers, the output layer (or output layers) of a network double as a handle to the network as a whole, so usually this is the only thing we will pass on to the rest of the code.

As mentioned above, `mnist.py` supports three types of models, and we implement that via three easily exchangeable functions of the same interface. First, we'll define a function that creates a Multi-Layer Perceptron (MLP) of a fixed architecture, explaining all the steps in detail. We'll then present a function generating an MLP of a custom architecture. Finally, we'll show how to create a Convolutional Neural Network (CNN).

Multi-Layer Perceptron (MLP)

The first function, `build_mlp()`, creates an MLP of two hidden layers of 800 units each, followed by a softmax output layer of 10 units. It applies 20% dropout to the input data and 50% dropout to the hidden layers. It is similar, but not fully equivalent to the smallest MLP in [Hinton2012] (that paper uses different nonlinearities, weight initialization and training).

The foundation of each neural network in Lasagne is an `InputLayer` instance (or multiple of those) representing the input data that will subsequently be fed to the network. Note that the `InputLayer` is not tied to any specific data yet, but only holds the shape of the data that will be passed to the network. In addition, it creates or can be linked to a Theano variable that will represent the network input in the Theano graph we'll build from the network later. Thus, our function starts like this:

```
def build_mlp(input_var=None):
    l_in = lasagne.layers.InputLayer(shape=(None, 1, 28, 28),
                                     input_var=input_var)
```

The four numbers in the shape tuple represent, in order: `(batchsize, channels, rows, columns)`. Here we've set the batchsize to `None`, which means the network will accept input data of arbitrary batchsize after compilation. If you know the batchsize beforehand and do not need this flexibility, you should give the batchsize here – especially for convolutional layers, this can allow Theano to apply some optimizations. `input_var` denotes the Theano variable we want to link the network's input layer to. If it is omitted (or set to `None`), the layer will just create a suitable variable itself, but it can be handy to link an existing variable to the network at construction time – especially if you're creating networks of multiple input layers. Here, we link it to a variable given as an argument to the `build_mlp()` function.

Before adding the first hidden layer, we'll apply 20% dropout to the input data. This is realized via a `DropoutLayer` instance:

```
l_in_drop = lasagne.layers.DropoutLayer(l_in, p=0.2)
```

Note that the first constructor argument is the incoming layer, such that `l_in_drop` is now stacked on top of `l_in`. All layers work this way, except for layers that merge multiple inputs: those accept a list of incoming layers as their first constructor argument instead.

We'll proceed with the first fully-connected hidden layer of 800 units. Note that when stacking a `DenseLayer` on higher-order input tensors, they will be flattened implicitly so we don't need to care about that. In this case, the input will be flattened from 1x28x28 images to 784-dimensional vectors.

```
l_hid1 = lasagne.layers.DenseLayer(
        l_in_drop, num_units=800,
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform())
```

Again, the first constructor argument means that we're stacking `l_hid1` on top of `l_in_drop`. `num_units` simply gives the number of units for this fully-connected layer. `nonlinearity` takes a nonlinearity function, several of which are defined in `lasagne.nonlinearities`. Here we've chosen the linear rectifier, so we'll obtain ReLUs. Finally, `lasagne.init.GlorotUniform()` gives the initializer for the weight matrix `W`. This particular initializer samples weights from a uniform distribution of a carefully chosen range. Other initializers are available in `lasagne.init`, and alternatively, `W` could also have been initialized from a Theano shared variable or numpy array of the correct shape (784x800 in this case, as the input to this layer has 1*28*28=784 dimensions). Note that `lasagne.init.GlorotUniform()` is the default, so we'll omit it from here – we just wanted to highlight that there is a choice.

We'll now add dropout of 50%, another 800-unit dense layer and 50% dropout again:

```
l_hid1_drop = lasagne.layers.DropoutLayer(l_hid1, p=0.5)

l_hid2 = lasagne.layers.DenseLayer(
        l_hid1_drop, num_units=800,
        nonlinearity=lasagne.nonlinearities.rectify)

l_hid2_drop = lasagne.layers.DropoutLayer(l_hid2, p=0.5)
```

Finally, we'll add the fully-connected output layer. The main difference is that it uses the softmax nonlinearity, as we're planning to solve a 10-class classification problem with this network.

```
l_out = lasagne.layers.DenseLayer(
        l_hid2_drop, num_units=10,
        nonlinearity=lasagne.nonlinearities.softmax)
```

As mentioned above, each layer is linked to its incoming layer(s), so we only need the output layer(s) to access a network in Lasagne:

```
return l_out
```

Custom MLP

The second function has a slightly more extensive signature:

```
def build_custom_mlp(input_var=None, depth=2, width=800, drop_input=.2,
                     drop_hidden=.5):
```

By default, it creates the same network as `build_mlp()` described above, but it can be customized with respect to the number and size of hidden layers, as well as the amount of input and hidden dropout. This demonstrates how creating a network in Python code can be a lot more flexible than a configuration file. See for yourself:

```
# Input layer and dropout (with shortcut `dropout` for `DropoutLayer`):
network = lasagne.layers.InputLayer(shape=(None, 1, 28, 28),
                                    input_var=input_var)
if drop_input:
    network = lasagne.layers.dropout(network, p=drop_input)
# Hidden layers and dropout:
nonlin = lasagne.nonlinearities.rectify
for _ in range(depth):
    network = lasagne.layers.DenseLayer(
```

```
            network, width, nonlinearity=nonlin)
    if drop_hidden:
        network = lasagne.layers.dropout(network, p=drop_hidden)
# Output layer:
softmax = lasagne.nonlinearities.softmax
network = lasagne.layers.DenseLayer(network, 10, nonlinearity=softmax)
return network
```

With two `if` clauses and a `for` loop, this network definition allows varying the architecture in a way that would be impossible for a `.yaml` file in Pylearn2 or a `.cfg` file in cuda-convnet.

Note that to make the code easier, all the layers are just called `network` here – there is no need to give them different names if all we return is the last one we created anyway; we just used different names before for clarity.

Convolutional Neural Network (CNN)

Finally, the `build_cnn()` function creates a CNN of two convolution and pooling stages, a fully-connected hidden layer and a fully-connected output layer. The function begins like the others:

```
def build_cnn(input_var=None):
    network = lasagne.layers.InputLayer(shape=(None, 1, 28, 28),
                                        input_var=input_var)
```

We don't apply dropout to the inputs, as this tends to work less well for convolutional layers. Instead of a `DenseLayer`, we now add a `Conv2DLayer` with 32 filters of size 5x5 on top:

```
network = lasagne.layers.Conv2DLayer(
        network, num_filters=32, filter_size=(5, 5),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform())
```

The nonlinearity and weight initializer can be given just as for the `DenseLayer` (and again, `GlorotUniform()` is the default, we'll omit it from now). Strided and padded convolutions are supported as well; see the `Conv2DLayer` docstring.

Note: For experts: `Conv2DLayer` will create a convolutional layer using `T.nnet.conv2d`, Theano's default convolution. On compilation for GPU, Theano replaces this with a cuDNN-based implementation if available, otherwise falls back to a gemm-based implementation. For details on this, please see the Theano convolution documentation.

Lasagne also provides convolutional layers directly enforcing a specific implementation: `lasagne.layers.dnn.Conv2DDNNLayer` to enforce cuDNN, `lasagne.layers.corrmm.Conv2DMMLayer` to enforce the gemm-based one, `lasagne.layers.cuda_convnet.Conv2DCCLayer` for Krizhevsky's cuda-convnet.

We then apply max-pooling of factor 2 in both dimensions, using a `MaxPool2DLayer` instance:

```
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))
```

We add another convolution and pooling stage like the ones before:

```
network = lasagne.layers.Conv2DLayer(
        network, num_filters=32, filter_size=(5, 5),
        nonlinearity=lasagne.nonlinearities.rectify)
network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))
```

Then a fully-connected layer of 256 units with 50% dropout on its inputs (using the `lasagne.layers.dropout` shortcut directly inline):

```
network = lasagne.layers.DenseLayer(
        lasagne.layers.dropout(network, p=.5),
        num_units=256,
        nonlinearity=lasagne.nonlinearities.rectify)
```

And finally a 10-unit softmax output layer, again with 50% dropout:

```
network = lasagne.layers.DenseLayer(
        lasagne.layers.dropout(network, p=.5),
        num_units=10,
        nonlinearity=lasagne.nonlinearities.rectify)

return network
```

Training the model

The remaining part of the `mnist.py` script copes with setting up and running a training loop over the MNIST dataset.

Dataset iteration

It first defines a short helper function for synchronously iterating over two numpy arrays of input data and targets, respectively, in mini-batches of a given number of items. For the purpose of this tutorial, we can shorten it to:

```
def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    if shuffle:
        ...
    for ...:
        yield inputs[...], targets[...]
```

All that's relevant is that it is a generator function that serves one batch of inputs and targets at a time until the given dataset (in `inputs` and `targets`) is exhausted, either in sequence or in random order. Below we will plug this function into our training loop, validation loop and test loop.

Preparation

Let's now focus on the `main()` function. A bit simplified, it begins like this:

```
# Load the dataset
X_train, y_train, X_val, y_val, X_test, y_test = load_dataset()
# Prepare Theano variables for inputs and targets
input_var = T.tensor4('inputs')
target_var = T.ivector('targets')
# Create neural network model
network = build_mlp(input_var)
```

The first line loads the inputs and targets of the MNIST dataset as numpy arrays, split into training, validation and test data. The next two statements define symbolic Theano variables that will represent a mini-batch of inputs and targets in all the Theano expressions we will generate for network training and inference. They are not tied to any data yet, but their dimensionality and data type is fixed already and matches the actual inputs and targets we will process later. Finally, we call one of the three functions for building the Lasagne network, depending on the first command line argument – we've just removed command line handling here for clarity. Note that we hand the symbolic input variable to `build_mlp()` so it will be linked to the network's input layer.

Loss and update expressions

Continuing, we create a loss expression to be minimized in training:

```
prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(prediction, target_var)
loss = loss.mean()
```

The first step generates a Theano expression for the network output given the input variable linked to the network's input layer(s). The second step defines a Theano expression for the categorical cross-entropy loss between said network output and the targets. Finally, as we need a scalar loss, we simply take the mean over the mini-batch. Depending on the problem you are solving, you will need different loss functions, see `lasagne.objectives` for more.

Having the model and the loss function defined, we create update expressions for training the network. An update expression describes how to change the trainable parameters of the network at each presented mini-batch. We will use Stochastic Gradient Descent (SGD) with Nesterov momentum here, but the `lasagne.updates` module offers several others you can plug in instead:

```
params = lasagne.layers.get_all_params(network, trainable=True)
updates = lasagne.updates.nesterov_momentum(
        loss, params, learning_rate=0.01, momentum=0.9)
```

The first step collects all Theano `SharedVariable` instances making up the trainable parameters of the layer, and the second step generates an update expression for each parameter.

For monitoring progress during training, after each epoch, we evaluate the network on the validation set. We need a slightly different loss expression for that:

```
test_prediction = lasagne.layers.get_output(network, deterministic=True)
test_loss = lasagne.objectives.categorical_crossentropy(test_prediction,
                                                        target_var)
test_loss = test_loss.mean()
```

The crucial difference is that we pass `deterministic=True` to the `get_output` call. This causes all nondeterministic layers to switch to a deterministic implementation, so in our case, it disables the dropout layers. As an additional monitoring quantity, we create an expression for the classification accuracy:

```
test_acc = T.mean(T.eq(T.argmax(test_prediction, axis=1), target_var),
                  dtype=theano.config.floatX)
```

It also builds on the deterministic `test_prediction` expression.

Compilation

Equipped with all the necessary Theano expressions, we're now ready to compile a function performing a training step:

```
train_fn = theano.function([input_var, target_var], loss, updates=updates)
```

This tells Theano to generate and compile a function taking two inputs – a mini-batch of images and a vector of corresponding targets – and returning a single output: the training loss. Additionally, each time it is invoked, it applies all parameter updates in the `updates` dictionary, thus performing a gradient descent step with Nesterov momentum.

For validation, we compile a second function:

```
val_fn = theano.function([input_var, target_var], [test_loss, test_acc])
```

This one also takes a mini-batch of images and targets, then returns the (deterministic) loss and classification accuracy, not performing any updates.

Training loop

We're finally ready to write the training loop. In essence, we just need to do the following:

```
for epoch in range(num_epochs):
    for batch in iterate_minibatches(X_train, y_train, 500, shuffle=True):
        inputs, targets = batch
        train_fn(inputs, targets)
```

This uses our dataset iteration helper function to iterate over the training data in random order, in mini-batches of 500 items each, for `num_epochs` epochs, and calls the training function we compiled to perform an update step of the network parameters.

But to be able to monitor the training progress, we capture the training loss, compute the validation loss and print some information to the console every time an epoch finishes:

```
for epoch in range(num_epochs):
    # In each epoch, we do a full pass over the training data:
    train_err = 0
    train_batches = 0
    start_time = time.time()
    for batch in iterate_minibatches(X_train, y_train, 500, shuffle=True):
        inputs, targets = batch
        train_err += train_fn(inputs, targets)
        train_batches += 1

    # And a full pass over the validation data:
    val_err = 0
    val_acc = 0
    val_batches = 0
    for batch in iterate_minibatches(X_val, y_val, 500, shuffle=False):
        inputs, targets = batch
        err, acc = val_fn(inputs, targets)
        val_err += err
        val_acc += acc
        val_batches += 1

    # Then we print the results for this epoch:
    print("Epoch {} of {} took {:.3f}s".format(
        epoch + 1, num_epochs, time.time() - start_time))
    print("  training loss:\t\t{:.6f}".format(train_err / train_batches))
    print("  validation loss:\t\t{:.6f}".format(val_err / val_batches))
    print("  validation accuracy:\t\t{:.2f} %".format(
        val_acc / val_batches * 100))
```

At the very end, we re-use the `val_fn()` function to compute the loss and accuracy on the test set, finishing the script.

1.2.4 Where to go from here

This finishes our introductory tutorial. For more information on what you can do with Lasagne's layers, just continue reading through *Layers* and *Creating custom layers*. More tutorials, examples and code snippets can be found in the Lasagne Recipes repository. Finally, the reference lists and explains all layers (`lasagne.layers`), weight initializers (`lasagne.init`), nonlinearities (`lasagne.nonlinearities`), loss expressions (`lasagne.objectives`), training methods (`lasagne.updates`) and regularizers (`lasagne.regularization`) included in the library, and should also make it simple to create your own.

1.3 Layers

The *lasagne.layers* module provides various classes representing the layers of a neural network. All of them are subclasses of the `lasagne.layers.Layer` base class.

1.3.1 Creating a layer

A layer can be created as an instance of a *Layer* subclass. For example, a dense layer can be created as follows:

```
>>> import lasagne
>>> l = lasagne.layers.DenseLayer(l_in, num_units=100)
```

This will create a dense layer with 100 units, connected to another layer *l_in*.

1.3.2 Creating a network

Note that for almost all types of layers, you will have to specify one or more other layers that the layer you are creating gets its input from. The main exception is `InputLayer`, which can be used to represent the input of a network.

Chaining layer instances together like this will allow you to specify your desired network structure. Note that the same layer can be used as input to multiple other layers, allowing for arbitrary tree and directed acyclic graph (DAG) structures.

Here is an example of an MLP with a single hidden layer:

```
>>> import theano.tensor as T
>>> l_in = lasagne.layers.InputLayer((100, 50))
>>> l_hidden = lasagne.layers.DenseLayer(l_in, num_units=200)
>>> l_out = lasagne.layers.DenseLayer(l_hidden, num_units=10,
...                                   nonlinearity=T.nnet.softmax)
```

The first layer of the network is an *InputLayer*, which represents the input. When creating an input layer, you should specify the shape of the input data. In this example, the input is a matrix with shape (100, 50), representing a batch of 100 data points, where each data point is a vector of length 50. The first dimension of a tensor is usually the batch dimension, following the established Theano and scikit-learn conventions.

The hidden layer of the network is a dense layer with 200 units, taking its input from the input layer. Note that we did not specify the nonlinearity of the hidden layer. A layer with rectified linear units will be created by default.

The output layer of the network is a dense layer with 10 units and a softmax nonlinearity, allowing for 10-way classification of the input vectors.

Note also that we did not create any object representing the entire network. Instead, the output layer instance *l_out* is also used to refer to the entire network in Lasagne.

1.3.3 Naming layers

For convenience, you can name a layer by specifying the *name* keyword argument:

```
>>> l_hidden = lasagne.layers.DenseLayer(l_in, num_units=200,
...                                      name="hidden_layer")
```

1.3.4 Initializing parameters

Many types of layers, such as `DenseLayer`, have trainable parameters. These are referred to by short names that match the conventions used in modern deep learning literature. For example, a weight matrix will usually be called *W*, and a bias vector will usually be *b*.

When creating a layer with trainable parameters, Theano shared variables will be created for them and initialized automatically. You can optionally specify your own initialization strategy by using keyword arguments that match the parameter variable names. For example:

```
>>> l = lasagne.layers.DenseLayer(l_in, num_units=100,
...                               W=lasagne.init.Normal(0.01))
```

The weight matrix *W* of this dense layer will be initialized using samples from a normal distribution with standard deviation 0.01 (see *lasagne.init* for more information).

There are several ways to manually initialize parameters:

- **Theano shared variable** If a shared variable instance is provided, this is used unchanged as the parameter variable. For example:

  ```
  >>> import theano
  >>> import numpy as np
  >>> W = theano.shared(np.random.normal(0, 0.01, (50, 100)))
  >>> l = lasagne.layers.DenseLayer(l_in, num_units=100, W=W)
  ```

- **numpy array** If a numpy array is provided, a shared variable is created and initialized using the array. For example:

  ```
  >>> W_init = np.random.normal(0, 0.01, (50, 100))
  >>> l = lasagne.layers.DenseLayer(l_in, num_units=100, W=W_init)
  ```

- **callable** If a callable is provided (e.g. a function or a `lasagne.init.Initializer` instance), a shared variable is created and the callable is called with the desired shape to generate suitable initial parameter values. The variable is then initialized with those values. For example:

  ```
  >>> l = lasagne.layers.DenseLayer(l_in, num_units=100,
  ...                               W=lasagne.init.Normal(0.01))
  ```

  Or, using a custom initialization function:

  ```
  >>> def init_W(shape):
  ...     return np.random.normal(0, 0.01, shape)
  >>> l = lasagne.layers.DenseLayer(l_in, num_units=100, W=init_W)
  ```

Some types of parameter variables can also be set to `None` at initialization (e.g. biases). In that case, the parameter variable will be omitted. For example, creating a dense layer without biases is done as follows:

```
>>> l = lasagne.layers.DenseLayer(l_in, num_units=100, b=None)
```

1.3.5 Parameter sharing

Parameter sharing between multiple layers can be achieved by using the same Theano shared variable instance for their parameters. For example:

```
>>> l1 = lasagne.layers.DenseLayer(l_in, num_units=100)
>>> l2 = lasagne.layers.DenseLayer(l_in, num_units=100, W=l1.W)
```

These two layers will now share weights (but have separate biases).

1.3.6 Propagating data through layers

To compute an expression for the output of a single layer given its input, the *get_output_for()* method can be used. To compute the output of a network, you should instead call `lasagne.layers.get_output()` on it. This will traverse the network graph.

You can call this function with the layer you want to compute the output expression for:

```
>>> y = lasagne.layers.get_output(l_out)
```

In that case, a Theano expression will be returned that represents the output in function of the input variables associated with the `lasagne.layers.InputLayer` instance (or instances) in the network, so given the example network from before, you could compile a Theano function to compute its output given an input as follows:

```
>>> f = theano.function([l_in.input_var], lasagne.layers.get_output(l_out))
```

You can also specify a Theano expression to use as input as a second argument to `lasagne.layers.get_output()`:

```
>>> x = T.matrix('x')
>>> y = lasagne.layers.get_output(l_out, x)
>>> f = theano.function([x], y)
```

This only works when there is only a single `InputLayer` in the network. If there is more than one, you can specify input expressions in a dictionary. For example, in a network with two input layers *l_in1* and *l_in2* and an output layer *l_out*:

```
>>> x1 = T.matrix('x1')
>>> x2 = T.matrix('x2')
>>> y = lasagne.layers.get_output(l_out, { l_in1: x1, l_in2: x2 })
```

Any keyword arguments passed to *get_output()* are propagated to all layers. This makes it possible to control the behavior of the entire network. The main use case for this is the `deterministic` keyword argument, which disables stochastic behaviour such as dropout when set to `True`. This is useful because a deterministic output is desirable at evaluation time.

```
>>> y = lasagne.layers.get_output(l_out, deterministic=True)
```

Some networks may have multiple output layers - or you may just want to compute output expressions for intermediate layers in the network. In that case, you can pass a list of layers. For example, in a network with two output layers *l_out1* and *l_out2*:

```
>>> y1, y2 = lasagne.layers.get_output([l_out1, l_out2])
```

You could also just call `lasagne.layers.get_output()` twice:

```
>>> y1 = lasagne.layers.get_output(l_out1)
>>> y2 = lasagne.layers.get_output(l_out2)
```

However, this is **not recommended**! Some network layers may have non-deterministic output, such as dropout layers. If you compute the network output expressions with separate calls to `lasagne.layers.get_output()`, they will not use the same samples. Furthermore, this may lead to unnecessary computation because Theano is not always able to merge identical computations properly. Calling *get_output()* only once prevents both of these issues.

1.4 Creating custom layers

1.4.1 A simple layer

To implement a custom layer in Lasagne, you will have to write a Python class that subclasses `Layer` and implement at least one method: *get_output_for()*. This method computes the output of the layer given its input. Note that both the output and the input are Theano expressions, so they are symbolic.

The following is an example implementation of a layer that multiplies its input by 2:

```
class DoubleLayer(lasagne.layers.Layer):
    def get_output_for(self, input, **kwargs):
        return 2 * input
```

This is all that's required to implement a functioning custom layer class in Lasagne.

1.4.2 A layer that changes the shape

If the layer does not change the shape of the data (for example because it applies an elementwise operation), then implementing only this one method is sufficient. Lasagne will assume that the output of the layer has the same shape as its input.

However, if the operation performed by the layer changes the shape of the data, you also need to implement *get_output_shape_for()*. This method computes the shape of the layer output given the shape of its input. Note that this shape computation should result in a tuple of integers, so it is *not* symbolic.

This method exists because Lasagne needs a way to propagate shape information when a network is defined, so it can determine what sizes the parameter tensors should be, for example. This mechanism allows each layer to obtain the size of its input from the previous layer, which means you don't have to specify the input size manually. This also prevents errors stemming from inconsistencies between the layers' expected and actual shapes.

We can implement a layer that computes the sum across the trailing axis of its input as follows:

```
class SumLayer(lasagne.layers.Layer):
    def get_output_for(self, input, **kwargs):
        return input.sum(axis=-1)

    def get_output_shape_for(self, input_shape):
        return input_shape[:-1]
```

It is important that the shape computation is correct, as this shape information may be used to initialize other layers in the network.

1.4.3 A layer with parameters

If the layer has parameters, these should be initialized in the constructor. In Lasagne, parameters are represented by Theano shared variables. A method is provided to create and register parameter variables: `lasagne.layers.Layer.add_param()`.

To show how this can be used, here is a layer that multiplies its input by a matrix `W` (much like a typical fully connected layer in a neural network would). This matrix is a parameter of the layer. The shape of the matrix will be `(num_inputs, num_units)`, where `num_inputs` is the number of input features and `num_units` has to be specified when the layer is created.

```
class DotLayer(lasagne.layers.Layer):
    def __init__(self, incoming, num_units, W=lasagne.init.Normal(0.01), **kwargs):
        super(DotLayer, self).__init__(incoming, **kwargs)
        num_inputs = self.input_shape[1]
        self.num_units = num_units
        self.W = self.add_param(W, (num_inputs, num_units), name='W')

    def get_output_for(self, input, **kwargs):
        return T.dot(input, self.W)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0], self.num_units)
```

A few things are worth noting here: when overriding the constructor, we need to call the superclass constructor on the first line. This is important to ensure the layer functions properly. Note that we pass `**kwargs` - although this is not strictly necessary, it enables some other cool Lasagne features, such as making it possible to give the layer a name:

```
>>> l_dot = DotLayer(l_in, num_units=50, name='my_dot_layer')
```

The call to `self.add_param()` creates the Theano shared variable representing the parameter, and registers it so it can later be retrieved using `lasagne.layers.Layer.get_params()`. It returns the created variable, which we tuck away in `self.W` for easy access.

Note that we've also made it possible to specify a custom initialization strategy for `W` by adding a constructor argument for it, e.g.:

```
>>> l_dot = DotLayer(l_in, num_units=50, W=lasagne.init.Constant(0.0))
```

This 'Lasagne idiom' of tucking away a created parameter variable in an attribute for easy access and adding a constructor argument with the same name to specify the initialization strategy is very common throughout the library.

Finally, note that we used `self.input_shape` to determine the shape of the parameter matrix. This property is available in all Lasagne layers, once the superclass constructor has been called.

1.4.4 A layer with multiple behaviors

Some layers can have multiple behaviors. For example, a layer implementing dropout should be able to be switched on or off. During training, we want it to apply dropout noise to its input and scale up the remaining values, but during evaluation we don't want it to do anything.

For this purpose, the *get_output_for()* method takes optional keyword arguments (`kwargs`). When *get_output()* is called to compute an expression for the output of a network, all specified keyword arguments are passed to the *get_output_for()* methods of all layers in the network.

For layers that add noise for regularization purposes, such as dropout, the convention in Lasagne is to use the keyword argument `deterministic` to control its behavior.

Lasagne's `lasagne.layers.DropoutLayer` looks roughly like this (simplified implementation for illustration purposes):

```
from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
_srng = RandomStreams()

class DropoutLayer(Layer):
    def __init__(self, incoming, p=0.5, **kwargs):
        super(DropoutLayer, self).__init__(incoming, **kwargs)
        self.p = p
```

```
    def get_output_for(self, input, deterministic=False, **kwargs):
        if deterministic:  # do nothing in the deterministic case
            return input
        else:  # add dropout noise otherwise
            retain_prob = 1 - self.p
            input /= retain_prob
            return input * _srng.binomial(input.shape, p=retain_prob,
                                          dtype=theano.config.floatX)
```

1.5 Development

The Lasagne project was started by Sander Dieleman in September 2014. It is developed by a core team of eight people (in alphabetical order: Eric Battenberg, Sander Dieleman, Daniel Nouri, Eben Olson, Aäron van den Oord, Colin Raffel, Jan Schlüter, Søren Kaae Sønderby) and numerous additional contributors on GitHub: https://github.com/Lasagne/Lasagne

As an open-source project by researchers for researchers, we highly welcome contributions! Every bit helps and will be credited.

1.5.1 Philosophy

Lasagne grew out of a need to combine the flexibility of Theano with the availability of the right building blocks for training neural networks. Its development is guided by a number of design goals:

- **Simplicity**: Be easy to use, easy to understand and easy to extend, to facilitate use in research. Interfaces should be kept small, with as few classes and methods as possible. Every added abstraction and feature should be carefully scrutinized, to determine whether the added complexity is justified.
- **Transparency**: Do not hide Theano behind abstractions, directly process and return Theano expressions or Python / numpy data types. Try to rely on Theano's functionality where possible, and follow Theano's conventions.
- **Modularity**: Allow all parts (layers, regularizers, optimizers, ...) to be used independently of Lasagne. Make it easy to use components in isolation or in conjunction with other frameworks.
- **Pragmatism**: Make common use cases easy, do not overrate uncommon cases. Ideally, everything should be possible, but common use cases shouldn't be made more difficult just to cater for exotic ones.
- **Restraint**: Do not obstruct users with features they decide not to use. Both in using and in extending components, it should be possible for users to be fully oblivious to features they do not need.
- **Focus**: "Do one thing and do it well". Do not try to provide a library for everything to do with deep learning.

1.5.2 What to contribute

Give feedback

To send us general feedback, questions or ideas for improvement, please post on our mailing list.

If you have a very concrete feature proposal, add it to the issue tracker on GitHub:

- Explain how it would work, and link to a scientific paper if applicable.
- Keep the scope as narrow as possible, to make it easier to implement.

Report bugs

Report bugs at the issue tracker on GitHub. If you are reporting a bug, please include:

- your Lasagne and Theano version.
- steps to reproduce the bug, ideally reduced to a few Python commands.
- the results you obtain, and the results you expected instead.

If you are unsure whether the behavior you experience is a bug, or if you are unsure whether it is related to Lasagne or Theano, please just ask on our mailing list first.

Fix bugs

Look through the GitHub issues for bug reports. Anything tagged with "bug" is open to whoever wants to implement it. If you discover a bug in Lasagne you can fix yourself, by all means feel free to just implement a fix and not report it first.

Implement features

Look through the GitHub issues for feature proposals. Anything tagged with "feature" or "enhancement" is open to whoever wants to implement it. If you have a feature in mind you want to implement yourself, please note that Lasagne has a fairly narrow focus and we strictly follow a set of *design principles*, so we cannot guarantee upfront that your code will be included. Please do not hesitate to just propose your idea in a GitHub issue or on the mailing list first, so we can discuss it and/or guide you through the implementation.

Write documentation

Whenever you find something not explained well, misleading, glossed over or just wrong, please update it! The *Edit on GitHub* link on the top right of every documentation page and the *[source]* link for every documented entity in the API reference will help you to quickly locate the origin of any text.

1.5.3 How to contribute

Edit on GitHub

As a very easy way of just fixing issues in the documentation, use the *Edit on GitHub* link on the top right of a documentation page or the *[source]* link of an entity in the API reference to open the corresponding source file in GitHub, then click the *Edit this file* link to edit the file in your browser and send us a Pull Request. All you need for this is a free GitHub account.

For any more substantial changes, please follow the steps below to setup Lasagne for development.

Development setup

First, follow the instructions for performing a development installation of Lasagne (including forking on GitHub): *Development installation*

To be able to run the tests and build the documentation locally, install additional requirements with: `pip install -r requirements-dev.txt` (adding `--user` if you want to install to your home directory instead).

If you use the bleeding-edge version of Theano, then instead of running that command, just use `pip install` to manually install all dependencies listed in `requirements-dev.txt` with their correct versions; otherwise it will attempt to downgrade Theano to the known good version in `requirements.txt`.

Documentation

The documentation is generated with Sphinx. To build it locally, run the following commands:

```
cd docs
make html
```

Afterwards, open `docs/_build/html/index.html` to view the documentation as it would appear on readthedocs. If you changed a lot and seem to get misleading error messages or warnings, run `make clean html` to force Sphinx to recreate all files from scratch.

When writing docstrings, follow existing documentation as much as possible to ensure consistency throughout the library. For additional information on the syntax and conventions used, please refer to the following documents:

- reStructuredText Primer
- Sphinx reST markup constructs
- A Guide to NumPy/SciPy Documentation

Testing

Lasagne has a code coverage of 100%, which has proven very helpful in the past, but also creates some duties:

- Whenever you change any code, you should test whether it breaks existing features by just running the test suite. The test suite will also be run by Travis for any Pull Request to Lasagne.
- Any code you add needs to be accompanied by tests ensuring that nobody else breaks it in future. Coveralls will check whether the code coverage stays at 100% for any Pull Request to Lasagne.
- Every bug you fix indicates a missing test case, so a proposed bug fix should come with a new test that fails without your fix.

To run the full test suite, just do

```
py.test
```

Testing will take over 5 minutes for the first run, but less than a minute for subsequent runs when Theano can reuse compiled code. It will end with a code coverage report specifying which code lines are not covered by tests, if any. Furthermore, it will list any failed tests, and failed PEP8 checks.

To only run tests matching a certain name pattern, use the `-k` command line switch, e.g., `-k pool` will run the pooling layer tests only.

To land in a `pdb` debug prompt on a failure to inspect it more closely, use the `--pdb` switch.

Finally, for a loop-on-failing mode, do `pip install pytest-xdist` and run `py.test -f`. This will pause after the run, wait for any source file to change and run all previously failing tests again.

Sending Pull Requests

When you're satisfied with your addition, the tests pass and the documentation looks good without any markup errors, commit your changes to a new branch, push that branch to your fork and send us a Pull Request via GitHub's web interface.

All these steps are nicely explained on GitHub: https://guides.github.com/introduction/flow/

When filing your Pull Request, please include a description of what it does, to help us reviewing it. If it is fixing an open issue, say, issue #123, add *Fixes #123*, *Resolves #123* or *Closes #123* to the description text, so GitHub will close it when your request is merged.

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 lasagne.layers

2.1.1 Helper functions

lasagne.layers.**get_output**(*layer_or_layers*, *inputs=None*, ***kwargs*)

Computes the output of the network at one or more given layers. Optionally, you can define the input(s) to propagate through the network instead of using the input variable(s) associated with the network's input layer(s).

Parameters **layer_or_layers** : Layer or list

the `Layer` instance for which to compute the output expressions, or a list of `Layer` instances.

inputs : None, Theano expression, numpy array, or dict

If None, uses the input variables associated with the `InputLayer` instances. If a Theano expression, this defines the input for a single `InputLayer` instance. Will throw a ValueError if there are multiple `InputLayer` instances. If a numpy array, this will be wrapped as a Theano constant and used just like a Theano expression. If a dictionary, any `Layer` instance (including the input layers) can be mapped to a Theano expression or numpy array to use instead of its regular output.

Returns **output** : Theano expression or list

the output of the given layer(s) for the given network input

Notes

Depending on your network architecture, *get_output([l1, l2])* may be crucially different from *[get_output(l1), get_output(l2)]*. Only the former ensures that the output expressions depend on the same intermediate expressions. For example, when *l1* and *l2* depend on a common dropout layer, the former will use the same dropout mask for both, while the latter will use two different dropout masks.

lasagne.layers.**get_output_shape**(*layer_or_layers*, *input_shapes=None*)

Computes the output shape of the network at one or more given layers.

Parameters **layer_or_layers** : Layer or list

the Layer instance for which to compute the output shapes, or a list of Layer instances.

input_shapes : None, tuple, or dict

If None, uses the input shapes associated with the InputLayer instances. If a tuple, this defines the input shape for a single InputLayer instance. Will throw a ValueError if there are multiple InputLayer instances. If a dictionary, any Layer instance (including the input layers) can be mapped to a shape tuple to use instead of its regular output shape.

Returns tuple or list

the output shape of the given layer(s) for the given network input

lasagne.layers.**get_all_layers**(*layer*, *treat_as_input=None*)

This function gathers all layers below one or more given Layer instances, including the given layer(s). Its main use is to collect all layers of a network just given the output layer(s). The layers are guaranteed to be returned in a topological order: a layer in the result list is always preceded by all layers its input depends on.

Parameters **layer** : Layer or list

the Layer instance for which to gather all layers feeding into it, or a list of Layer instances.

treat_as_input : None or iterable

an iterable of Layer instances to treat as input layers with no layers feeding into them. They will show up in the result list, but their incoming layers will not be collected (unless they are required for other layers as well).

Returns list

a list of Layer instances feeding into the given instance(s) either directly or indirectly, and the given instance(s) themselves, in topological order.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> get_all_layers(l1) == [l_in, l1]
True
>>> l2 = DenseLayer(l_in, num_units=10)
>>> get_all_layers([l2, l1]) == [l_in, l2, l1]
True
>>> get_all_layers([l1, l2]) == [l_in, l1, l2]
True
>>> l3 = DenseLayer(l2, num_units=20)
>>> get_all_layers(l3) == [l_in, l2, l3]
True
>>> get_all_layers(l3, treat_as_input=[l2]) == [l2, l3]
True
```

lasagne.layers.**get_all_params**(*layer*, ***tags*)

This function gathers all parameters of all layers below one or more given Layer instances, including the layer(s) itself. Its main use is to collect all parameters of a network just given the output layer(s).

By default, all parameters that participate in the forward pass will be returned. The list can optionally be filtered by specifying tags as keyword arguments. For example, trainable=True will only return trainable

parameters, and `regularizable=True` will only return parameters that can be regularized (e.g., by L2 decay).

Parameters **layer** : Layer or list

The `Layer` instance for which to gather all parameters, or a list of `Layer` instances.

****tags (optional)**

tags can be specified to filter the list. Specifying `tag1=True` will limit the list to parameters that are tagged with `tag1`. Specifying `tag1=False` will limit the list to parameters that are not tagged with `tag1`. Commonly used tags are `regularizable` and `trainable`.

Returns **params** : list

A list of Theano shared variables representing the parameters.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> all_params = get_all_params(l1)
>>> all_params == [l1.W, l1.b]
True
```

`lasagne.layers.`**`count_params`**(*layer*, ***tags*)

This function counts all parameters (i.e., the number of scalar values) of all layers below one or more given `Layer` instances, including the layer(s) itself.

This is useful to compare the capacity of various network architectures. All parameters returned by the `Layer`'s' `get_params` methods are counted.

Parameters **layer** : Layer or list

The `Layer` instance for which to count the parameters, or a list of `Layer` instances.

****tags (optional)**

tags can be specified to filter the list of parameter variables that will be included in the count. Specifying `tag1=True` will limit the list to parameters that are tagged with `tag1`. Specifying `tag1=False` will limit the list to parameters that are not tagged with `tag1`. Commonly used tags are `regularizable` and `trainable`.

Returns int

The total number of learnable parameters.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> param_count = count_params(l1)
>>> param_count
1050
>>> param_count == 20 * 50 + 50  # 20 input * 50 units + 50 biases
True
```

lasagne.layers.**get_all_param_values**(*layer*, ***tags*)

This function returns the values of the parameters of all layers below one or more given Layer instances, including the layer(s) itself.

This function can be used in conjunction with set_all_param_values to save and restore model parameters.

Parameters **layer** : Layer or list

The Layer instance for which to gather all parameter values, or a list of Layer instances.

****tags (optional)**

tags can be specified to filter the list. Specifying tag1=True will limit the list to parameters that are tagged with tag1. Specifying tag1=False will limit the list to parameters that are not tagged with tag1. Commonly used tags are regularizable and trainable.

Returns list of numpy.array

A list of numpy arrays representing the parameter values.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> all_param_values = get_all_param_values(l1)
>>> (all_param_values[0] == l1.W.get_value()).all()
True
>>> (all_param_values[1] == l1.b.get_value()).all()
True
```

lasagne.layers.**set_all_param_values**(*layer*, *values*, ***tags*)

Given a list of numpy arrays, this function sets the parameters of all layers below one or more given Layer instances (including the layer(s) itself) to the given values.

This function can be used in conjunction with get_all_param_values to save and restore model parameters.

Parameters **layer** : Layer or list

The Layer instance for which to set all parameter values, or a list of Layer instances.

values : list of numpy.array

A list of numpy arrays representing the parameter values, must match the number of parameters. Every parameter's shape must match the shape of its new value.

****tags (optional)**

tags can be specified to filter the list of parameters to be set. Specifying tag1=True will limit the list to parameters that are tagged with tag1. Specifying tag1=False will limit the list to parameters that are not tagged with tag1. Commonly used tags are regularizable and trainable.

Raises **ValueError**

If the number of values is not equal to the number of params, or if a parameter's shape does not match the shape of its new value.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> all_param_values = get_all_param_values(l1)
>>> # all_param_values is now [l1.W.get_value(), l1.b.get_value()]
>>> # ...
>>> set_all_param_values(l1, all_param_values)
>>> # the parameter values are restored.
```

2.1.2 Layer base classes

class `lasagne.layers.`**`Layer`**(*incoming*, *name=None*)

The `Layer` class represents a single layer of a neural network. It should be subclassed when implementing new types of layers.

Because each layer can keep track of the layer(s) feeding into it, a network's output `Layer` instance can double as a handle to the full network.

`add_param`(*spec*, *shape*, *name=None*, ***tags*)

Register and initialize a Theano shared variable containing parameters associated with the layer.

When defining a new layer, this method can be used in the constructor to define which parameters the layer has, what their shapes are, how they should be initialized and what tags are associated with them.

All parameter variables associated with the layer can be retrieved using `Layer.get_params()`.

Parameters **spec** : Theano shared variable, numpy array or callable

an initializer for this parameter variable. This should initialize the variable with an array of the specified shape. See `lasagne.utils.create_param()` for more information.

shape : tuple of int

a tuple of integers representing the desired shape of the parameter array.

name : str (optional)

the name of the parameter variable. This will be passed to `theano.shared` when the variable is created. If `spec` is already a shared variable, this parameter will be ignored to avoid overwriting an existing name. If the layer itself has a name, the name of the parameter variable will be prefixed with it and it will be of the form 'layer_name.param_name'.

****tags (optional)**

tags associated with the parameter variable can be specified as keyword arguments.

To associate the tag `tag1` with the variable, pass `tag1=True`.

By default, the tags `regularizable` and `trainable` are associated with the parameter variable. Pass `regularizable=False` or `trainable=False` respectively to prevent this.

Returns Theano shared variable

the resulting parameter variable

Notes

It is recommend to assign the resulting parameter variable to an attribute of the layer, so it can be accessed easily, for example:

```
>>> self.W = self.add_param(W, (2, 3), name='W')
```

get_output_for(*input*, ***kwargs*)
Propagates the given input through this layer (and only this layer).

Parameters **input** : Theano expression

The expression to propagate through this layer.

Returns **output** : Theano expression

The output of this layer given the input to this layer.

Notes

This is called by the base `lasagne.layers.get_output()` to propagate data through a network.

This method should be overridden when implementing a new `Layer` class. By default it raises *NotImplementedError*.

get_output_shape_for(*input_shape*)
Computes the output shape of this layer, given an input shape.

Parameters **input_shape** : tuple

A tuple representing the shape of the input. The tuple should have as many elements as there are input dimensions, and the elements should be integers or *None*.

Returns tuple

A tuple representing the shape of the output of this layer. The tuple has as many elements as there are output dimensions, and the elements are all either integers or *None*.

Notes

This method will typically be overridden when implementing a new `Layer` class. By default it simply returns the input shape. This means that a layer that does not modify the shape (e.g. because it applies an elementwise operation) does not need to override this method.

get_params(***tags*)
Returns a list of all the Theano variables that parameterize the layer.

By default, all parameters that participate in the forward pass will be returned (in the order they were registered in the Layer's constructor via `add_param()`). The list can optionally be filtered by specifying tags as keyword arguments. For example, `trainable=True` will only return trainable parameters, and `regularizable=True` will only return parameters that can be regularized (e.g., by L2 decay).

Parameters ****tags (optional)**

tags can be specified to filter the list. Specifying `tag1=True` will limit the list to parameters that are tagged with `tag1`. Specifying `tag1=False` will limit the list to parameters that are not tagged with `tag1`. Commonly used tags are `regularizable` and `trainable`.

Returns list of Theano shared variables

A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class lasagne.layers.**MergeLayer**(*incomings*, *name=None*)

This class represents a layer that aggregates input from multiple layers. It should be subclassed when implementing new types of layers that obtain their input from multiple layers.

get_output_for(*inputs*, ***kwargs*)

Propagates the given inputs through this layer (and only this layer).

Parameters **inputs** : list of Theano expressions

The Theano expressions to propagate through this layer.

Returns Theano expressions

The output of this layer given the inputs to this layer.

Notes

This is called by the base lasagne.layers.get_output() to propagate data through a network.

This method should be overridden when implementing a new Layer class with multiple inputs. By default it raises *NotImplementedError*.

get_output_shape_for(*input_shapes*)

Computes the output shape of this layer, given a list of input shapes.

Parameters **input_shape** : list of tuple

A list of tuples, with each tuple representing the shape of one of the inputs (in the correct order). These tuples should have as many elements as there are input dimensions, and the elements should be integers or *None*.

Returns tuple

A tuple representing the shape of the output of this layer. The tuple has as many elements as there are output dimensions, and the elements are all either integers or *None*.

Notes

This method must be overridden when implementing a new Layer class with multiple inputs. By default it raises *NotImplementedError*.

2.1.3 Network input

class lasagne.layers.**InputLayer**(*shape*, *input_var=None*, *name=None*, ***kwargs*)

This layer holds a symbolic variable that represents a network input. A variable can be specified when the layer is instantiated, else it is created.

Parameters **shape** : tuple of *int* or *None* elements

The shape of the input. Any element can be *None* to indicate that the size of that dimension is not fixed at compile time.

input_var : Theano symbolic variable or *None* (default: *None*)

A variable representing a network input. If it is not provided, a variable will be created.

Raises **ValueError**

If the dimension of *input_var* is not equal to *len(shape)*

Notes

The first dimension usually indicates the batch size. If you specify it, Theano may apply more optimizations while compiling the training or prediction function, but the compiled function will not accept data of a different batch size at runtime. To compile for a variable batch size, set the first shape element to *None* instead.

Examples

```
>>> from lasagne.layers import InputLayer
>>> l_in = InputLayer((100, 20))
```

2.1.4 Dense layers

class `lasagne.layers.`**`DenseLayer`**(*incoming*, *num_units*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, ***kwargs*)

A fully connected layer.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape

num_units : int

The number of units of the layer

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. If a shared variable or a numpy array is provided the shape should be (num_inputs, num_units). See `Layer.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If a shared variable or a numpy array is provided the shape should be (num_units,). If None is provided the layer will have no biases. See `Layer.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

Notes

If the input to this layer has more than two axes, it will flatten the trailing axes. This is useful for when a dense layer follows a convolutional layer, for example. It is not necessary to insert a `FlattenLayer` in this case.

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
```

class lasagne.layers.**NonlinearityLayer**(*incoming*, *nonlinearity=lasagne.nonlinearities.rectify*, ***kwargs*)

A layer that just applies a nonlinearity.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

class lasagne.layers.**NINLayer**(*incoming*, *num_units*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, ***kwargs*)

Network-in-network layer. Like DenseLayer, but broadcasting across all trailing dimensions beyond the 2nd. This results in a convolution operation with filter size 1 on all trailing dimensions. Any number of trailing dimensions is supported, so NINLayer can be used to implement 1D, 2D, 3D, ... convolutions.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape

num_units : int

The number of units of the layer

untie_biases : bool

If false the network has a single bias vector similar to a dense layer. If true a separate bias vector is used for each trailing dimension beyond the 2nd.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. If a shared variable or a numpy array is provided the shape should be (num_inputs, num_units), where num_units is the size of the 2nd. dimension of the input. See lasagne.utils.create_param() for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If a shared variable or a numpy array is provided the correct shape is determined by the untie_biases setting. If untie_biases is False, then the shape should be (num_units,). If untie_biases is True then the shape should be (num_units, input_dim[2], ..., input_dim[-1]). If None is provided the layer will have no biases. See lasagne.utils.create_param() for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

References

[R14]

Examples

```
>>> from lasagne.layers import InputLayer, NINLayer
>>> l_in = InputLayer((100, 20, 10, 3))
>>> l1 = NINLayer(l_in, num_units=5)
```

2.1.5 Convolutional layers

class lasagne.layers.**Conv1DLayer**(*incoming*, *num_filters*, *filter_size*, *stride=1*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *convolution=lasagne.theano_extensions.conv.conv1d_mc0*, ***kwargs*)

1D convolutional layer

Performs a 1D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape. The output of this layer should be a 3D tensor, with shape `(batch_size, num_input_channels, input_length)`.

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 1-element tuple specifying the size of the filters.

stride : int or iterable of int

An integer or a 1-element tuple specifying the stride of the convolution operation.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

An integer or a 1-element tuple results in symmetric zero-padding of the given size on both borders.

`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size on both sides (one less on the second side for an even filter size). When `stride=1`, this results in an output size equal to the input size.

`'valid'` is an alias for `0` (no padding / a valid convolution).

untie_biases : bool (default: False)

If `False`, the layer will have a bias parameter for each channel, which is shared across all positions in this channel. As a result, the *b* attribute will be a vector (1D).

If True, the layer will have separate bias parameters for each position in each channel. As a result, the *b* attribute will be a matrix (2D).

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. This should initialize the layer weights to a 3D array with shape `(num_filters, num_input_channels, filter_length)`. See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If None is provided, the layer will have no biases. This should initialize the layer biases to a 1D array with shape `(num_filters,)` if *untied_biases* is set to `False`. If it is set to `True`, its shape should be `(num_filters, input_length)` instead. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

convolution : callable

The convolution implementation to use. The *lasagne.theano_extensions.conv* module provides some alternative implementations for 1D convolutions, because the Theano API only features a 2D convolution implementation. Usually it should be fine to leave this at the default value.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

Notes

Theano's underlying convolution (`theano.tensor.nnet.conv.conv2d()`) only supports `pad=0` and `pad='full'`. This layer emulates other modes by cropping a full convolution or explicitly padding the input with zeros.

Attributes

W	(Theano shared variable) Variable representing the filter weights.
b	(Theano shared variable) Variable representing the biases.

get_W_shape()
Get the shape of the weight matrix *W*.

Returns tuple of int

The shape of the weight matrix.

class `lasagne.layers.`**Conv2DLayer**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *convolution=theano.tensor.nnet.conv2d*, ***kwargs*)
2D convolutional layer

Performs a 2D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape. The output of this layer should be a 4D tensor, with shape `(batch_size, num_input_channels, input_rows, input_columns)`.

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 2-element tuple specifying the size of the filters.

stride : int or iterable of int

An integer or a 2-element tuple specifying the stride of the convolution operation.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

A single integer results in symmetric zero-padding of the given size on all borders, a tuple of two integers allows different symmetric padding per dimension.

`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size on both sides (one less on the second side for an even filter size). When `stride=1`, this results in an output size equal to the input size.

`'valid'` is an alias for `0` (no padding / a valid convolution).

Note that `'full'` and `'same'` can be faster than equivalent integer values due to optimizations by Theano.

untie_biases : bool (default: False)

If `False`, the layer will have a bias parameter for each channel, which is shared across all positions in this channel. As a result, the *b* attribute will be a vector (1D).

If True, the layer will have separate bias parameters for each position in each channel. As a result, the *b* attribute will be a 3D tensor.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. This should initialize the layer weights to a 4D array with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If None is provided, the layer will have no biases. This should initialize the layer biases to a 1D array with shape `(num_filters,)` if *untied_biases* is set to `False`. If it is set to `True`, its shape should be `(num_filters, input_rows, input_columns)` instead. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

convolution : callable

The convolution implementation to use. Usually it should be fine to leave this at the default value.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

Notes

Theano's underlying convolution (`theano.tensor.nnet.conv.conv2d()`) only supports `pad=0` and `pad='full'`. This layer emulates other modes by cropping a full convolution or explicitly padding the input with zeros.

Attributes

W	(Theano shared variable) Variable representing the filter weights.
b	(Theano shared variable) Variable representing the biases.

get_W_shape()

Get the shape of the weight matrix *W*.

Returns tuple of int

The shape of the weight matrix.

Note: For experts: `Conv2DLayer` will create a convolutional layer using `T.nnet.conv2d`, Theano's default convolution. On compilation for GPU, Theano replaces this with a cuDNN-based implementation if available, otherwise falls back to a gemm-based implementation. For details on this, please see the Theano convolution documentation.

Lasagne also provides convolutional layers directly enforcing a specific implementation: `lasagne.layers.dnn.Conv2DDNNLayer` to enforce cuDNN, `lasagne.layers.corrmm.Conv2DMMLayer` to enforce the gemm-based one, `lasagne.layers.cuda_convnet.Conv2DCCLayer` for Krizhevsky's cuda-convnet.

2.1.6 Pooling layers

class `lasagne.layers.`**MaxPool1DLayer**(*incoming*, *pool_size*, *stride=None*, *pad=0*, *ignore_border=True*, ***kwargs*)

1D max-pooling layer

Performs 1D max-pooling over the trailing axis of a 3D input tensor.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer or iterable

The length of the pooling region. If an iterable, it should have a single element.

stride : integer, iterable or `None`

The stride between sucessive pooling regions. If `None` then `stride == pool_size`.

pad : integer or iterable

The number of elements to be added to the input on each side. Must be less than stride.

ignore_border : bool

If `True`, partial pooling regions will be ignored. Must be `True` if `pad != 0`.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

Notes

The value used to pad the input is chosen to be less than the minimum of the input, so that the output of each pooling region always corresponds to some element in the unpadded input region.

Using `ignore_border=False` prevents Theano from using cuDNN for the operation, so it will fall back to a slower implementation.

class `lasagne.layers.`**`MaxPool2DLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0)*, *ignore_border=True*, ***kwargs*)

2D max-pooling layer

Performs 2D max-pooling over the two trailing axes of a 4D input tensor.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer or iterable

The length of the pooling region in each dimension. If an integer, it is promoted to a square pooling region. If an iterable, it should have two elements.

stride : integer, iterable or `None`

The strides between sucessive pooling regions in each dimension. If `None` then `stride = pool_size`.

pad : integer or iterable

Number of elements to be added on each side of the input in each dimension. Each value must be less than the corresponding stride.

ignore_border : bool

If `True`, partial pooling regions will be ignored. Must be `True` if `pad != (0, 0)`.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

Notes

The value used to pad the input is chosen to be less than the minimum of the input, so that the output of each pooling region always corresponds to some element in the unpadded input region.

Using `ignore_border=False` prevents Theano from using cuDNN for the operation, so it will fall back to a slower implementation.

class `lasagne.layers.`**`Pool2DLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0)*, *ignore_border=True*, *mode='max'*, ***kwargs*)

2D pooling layer

Performs 2D mean or max-pooling over the two trailing axes of a 4D input tensor.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer or iterable

The length of the pooling region in each dimension. If an integer, it is promoted to a square pooling region. If an iterable, it should have two elements.

stride : integer, iterable or `None`

The strides between sucessive pooling regions in each dimension. If `None` then `stride = pool_size`.

pad : integer or iterable

Number of elements to be added on each side of the input in each dimension. Each value must be less than the corresponding stride.

ignore_border : bool

If `True`, partial pooling regions will be ignored. Must be `True` if `pad != (0, 0)`.

mode : {'max', 'average_inc_pad', 'average_exc_pad'}

Pooling mode: max-pooling or mean-pooling including/excluding zeros from partially padded pooling regions. Default is 'max'.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

See also:

`MaxPool2DLayer` Shortcut for max pooling layer.

Notes

The value used to pad the input is chosen to be less than the minimum of the input, so that the output of each pooling region always corresponds to some element in the unpadded input region.

Using `ignore_border=False` prevents Theano from using cuDNN for the operation, so it will fall back to a slower implementation.

class `lasagne.layers.`**`GlobalPoolLayer`**(*incoming*, *pool_function=theano.tensor.mean*, ***kwargs*)
Global pooling layer

This layer pools globally across all trailing dimensions beyond the 2nd.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_function : callable

the pooling function to use. This defaults to *theano.tensor.mean* (i.e. mean-pooling) and can be replaced by any other aggregation function.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

class lasagne.layers.**FeaturePoolLayer**(*incoming*, *pool_size*, *axis=1*, *pool_function=theano.tensor.max*, ***kwargs*)

Feature pooling layer

This layer pools across a given axis of the input. By default this is axis 1, which corresponds to the feature axis for DenseLayer, Conv1DLayer and Conv2DLayer. The layer can be used to implement maxout.

Parameters **incoming** : a Layer instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer

the size of the pooling regions, i.e. the number of features / feature maps to be pooled together.

axis : integer

the axis along which to pool. The default value of 1 works for DenseLayer, Conv1DLayer and Conv2DLayer.

pool_function : callable

the pooling function to use. This defaults to *theano.tensor.max* (i.e. max-pooling) and can be replaced by any other aggregation function.

****kwargs**

Any additional keyword arguments are passed to the Layer superclass.

Notes

This layer requires that the size of the axis along which it pools is a multiple of the pool size.

class lasagne.layers.**FeatureWTALayer**(*incoming*, *pool_size*, *axis=1*, ***kwargs*)

'Winner Take All' layer

This layer performs 'Winner Take All' (WTA) across feature maps: zero out all but the maximal activation value within a region.

Parameters **incoming** : a Layer instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer

the number of feature maps per region.

axis : integer

the axis along which the regions are formed.

****kwargs**

Any additional keyword arguments are passed to the Layer superclass.

Notes

This layer requires that the size of the axis along which it groups units is a multiple of the pool size.

2.1.7 Recurrent layers

Layers to construct recurrent networks. Recurrent layers can be used similarly to feed-forward layers except that the input shape is expected to be `(batch_size, sequence_length, num_inputs)`. The CustomRecurrentLayer can also support more than one "feature" dimension (e.g. using convolutional connections), but for all other layers, dimensions trailing the third dimension are flattened.

The following recurrent layers are implemented:

`CustomRecurrentLayer`	A layer which implements a recurrent connection.
`RecurrentLayer`	Dense recurrent neural network (RNN) layer
`LSTMLayer`	A long short-term memory (LSTM) layer.
`GRULayer`	Gated Recurrent Unit (GRU) Layer

For recurrent layers with gates we use a helper class to set up the parameters in each gate:

`Gate`	Simple class to hold the parameters for a gate connection.

Please refer to that class if you need to modify initial conditions of gates.

Recurrent layers and feed-forward layers can be combined in the same network by using a few reshape operations; please refer to the example below.

Examples

The following example demonstrates how recurrent layers can be easily mixed with feed-forward layers using `ReshapeLayer` and how to build a network with variable batch size and number of time steps.

```
>>> from lasagne.layers import *
>>> num_inputs, num_units, num_classes = 10, 12, 5
>>> # By setting the first two dimensions as None, we are allowing them to vary
>>> # They correspond to batch size and sequence length, so we will be able to
>>> # feed in batches of varying size with sequences of varying length.
>>> l_inp = InputLayer((None, None, num_inputs))
>>> # We can retrieve symbolic references to the input variable's shape, which
>>> # we will later use in reshape layers.
>>> batchsize, seqlen, _ = l_inp.input_var.shape
>>> l_lstm = LSTMLayer(l_inp, num_units=num_units)
>>> # In order to connect a recurrent layer to a dense layer, we need to
>>> # flatten the first two dimensions (our "sample dimensions"); this will
>>> # cause each time step of each sequence to be processed independently
>>> l_shp = ReshapeLayer(l_lstm, (-1, num_units))
>>> l_dense = DenseLayer(l_shp, num_units=num_classes)
>>> # To reshape back to our original shape, we can use the symbolic shape
>>> # variables we retrieved above.
>>> l_out = ReshapeLayer(l_dense, (batchsize, seqlen, num_classes))
```

class `lasagne.layers.`**`CustomRecurrentLayer`**(*incoming*, *input_to_hidden*, *hidden_to_hidden*, *nonlinearity=lasagne.nonlinearities.rectify*, *hid_init=lasagne.init.Constant(0.)*, *backwards=False*, *learn_init=False*, *gradient_steps=-1*, *grad_clipping=False*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, ***kwargs*)

A layer which implements a recurrent connection.

This layer allows you to specify custom input-to-hidden and hidden-to-hidden connections by instantiating `lasagne.layers.Layer` instances and passing them on initialization. Note that these connections can consist of multiple layers chained together. The output shape for the provided input-to-hidden and hidden-to-hidden connections must be the same. If you are looking for a standard, densely-connected recurrent layer, please see `RecurrentLayer`. The output is computed by

$$h_t = \sigma(f_i(x_t) + f_h(h_{t-1}))$$

Parameters **incoming** : a `lasagne.layers.Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape.

input_to_hidden : `lasagne.layers.Layer`

`lasagne.layers.Layer` instance which connects input to the hidden state (f_i). This layer may be connected to a chain of layers, which must end in a `lasagne.layers.InputLayer` with the same input shape as *incoming*.

hidden_to_hidden : `lasagne.layers.Layer`

Layer which connects the previous hidden state to the new state (f_h). This layer may be connected to a chain of layers, which must end in a `lasagne.layers.InputLayer` with the same input shape as *hidden_to_hidden*'s output shape.

nonlinearity : callable or None

Nonlinearity to apply when computing new state (σ). If None is provided, no nonlinearity will be applied.

hid_init : callable, np.ndarray, theano.shared or TensorVariable

Initializer for initial hidden state (h_0). If a TensorVariable (Theano expression) is supplied, it will not be learned regardless of the value of *learn_init*.

backwards : bool

If True, process the sequence backwards and then reverse the output again such that the output from the layer is always from x_1 to x_n.

learn_init : bool

If True, initial hidden values are learned. If *hid_init* is a TensorVariable then the TensorVariable is used and *learn_init* is ignored.

gradient_steps : int

Number of timesteps to include in the backpropagated gradient. If -1, backpropagate through the entire sequence.

grad_clipping : False or float

If a float is provided, the gradient messages are clipped during the backward pass. If False, the gradients will not be clipped. See [R25] (p. 6) for further explanation.

unroll_scan : bool

If True the recursion is unrolled instead of using scan. For some graphs this gives a significant speed up but it might also consume more memory. When *unroll_scan* is True, backpropagation always includes the full sequence, so *gradient_steps* must be set to -1 and the input sequence length must be known at compile time (i.e., cannot be given as None).

precompute_input : bool

If True, precompute input_to_hid before iterating through the sequence. This can result in a speedup at the expense of an increase in memory usage.

mask_input : `lasagne.layers.Layer`

Layer which allows for a sequence mask to be input, for when sequences are of variable length. Default *None*, which means no mask will be supplied (i.e. all sequences are of the same length).

References

[R25]

Examples

The following example constructs a simple *CustomRecurrentLayer* which has dense input-to-hidden and hidden-to-hidden connections.

```
>>> import lasagne
>>> n_batch, n_steps, n_in = (2, 3, 4)
>>> n_hid = 5
>>> l_in = lasagne.layers.InputLayer((n_batch, n_steps, n_in))
>>> l_in_hid = lasagne.layers.DenseLayer(
...     lasagne.layers.InputLayer((None, n_in)), n_hid)
>>> l_hid_hid = lasagne.layers.DenseLayer(
...     lasagne.layers.InputLayer((None, n_hid)), n_hid)
>>> l_rec = lasagne.layers.CustomRecurrentLayer(l_in, l_in_hid, l_hid_hid)
```

The CustomRecurrentLayer can also support "convolutional recurrence", as is demonstrated below.

```
>>> n_batch, n_steps, n_channels, width, height = (2, 3, 4, 5, 6)
>>> n_out_filters = 7
>>> filter_shape = (3, 3)
>>> l_in = lasagne.layers.InputLayer(
...     (n_batch, n_steps, n_channels, width, height))
>>> l_in_to_hid = lasagne.layers.Conv2DLayer(
...     lasagne.layers.InputLayer((None, n_channels, width, height)),
...     n_out_filters, filter_shape, pad='same')
>>> l_hid_to_hid = lasagne.layers.Conv2DLayer(
...     lasagne.layers.InputLayer(l_in_to_hid.output_shape),
...     n_out_filters, filter_shape, pad='same')
>>> l_rec = lasagne.layers.CustomRecurrentLayer(
...     l_in, l_in_to_hid, l_hid_to_hid)
```

get_output_for(*inputs*, ***kwargs*)

Compute this layer's output function given a symbolic input variable.

Parameters **inputs** : list of theano.TensorType

inputs[0] should always be the symbolic input variable. When this layer has a mask input (i.e. was instantiated with *mask_input != None*, indicating that the lengths of sequences in each batch vary), *inputs* should have length 2, where *inputs[1]* is the *mask*. The *mask* should be supplied as a Theano variable denoting whether each time step in each sequence in the batch is part of the sequence or not. *mask* should be a matrix of shape `(n_batch, n_time_steps)` where `mask[i, j] = 1` when `j <= (length of sequence i)` and `mask[i, j] = 0` when `j > (length of sequence i)`.

Returns **layer_output** : theano.TensorType

Symbolic output variable.

class `lasagne.layers.`**`RecurrentLayer`**(*incoming*, *num_units*, *W_in_to_hid=lasagne.init.Uniform()*, *W_hid_to_hid=lasagne.init.Uniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *hid_init=lasagne.init.Constant(0.)*, *backwards=False*, *learn_init=False*, *gradient_steps=-1*, *grad_clipping=False*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, ***kwargs*)

Dense recurrent neural network (RNN) layer

A "vanilla" RNN layer, which has dense input-to-hidden and hidden-to-hidden connections. The output is computed as

$$h_t = \sigma(x_t W_x + h_{t-1} W_h + b)$$

Parameters **incoming** : a `lasagne.layers.Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape.

num_units : int

Number of hidden units in the layer.

W_in_to_hid : Theano shared variable, numpy array or callable

Initializer for input-to-hidden weight matrix (W_x).

W_hid_to_hid : Theano shared variable, numpy array or callable

Initializer for hidden-to-hidden weight matrix (W_h).

b : Theano shared variable, numpy array, callable or None

Initializer for bias vector (b). If None is provided there will be no bias.

nonlinearity : callable or None

Nonlinearity to apply when computing new state (σ). If None is provided, no nonlinearity will be applied.

hid_init : callable, np.ndarray, theano.shared or TensorVariable

Initializer for initial hidden state (h_0). If a TensorVariable (Theano expression) is supplied, it will not be learned regardless of the value of *learn_init*.

backwards : bool

If True, process the sequence backwards and then reverse the output again such that the output from the layer is always from x_1 to x_n.

learn_init : bool

If True, initial hidden values are learned. If *hid_init* is a TensorVariable then *learn_init* is ignored.

gradient_steps : int

Number of timesteps to include in the backpropagated gradient. If -1, backpropagate through the entire sequence.

grad_clipping : False or float

If a float is provided, the gradient messages are clipped during the backward pass. If False, the gradients will not be clipped. See [R26] (p. 6) for further explanation.

unroll_scan : bool

If True the recursion is unrolled instead of using scan. For some graphs this gives a significant speed up but it might also consume more memory. When *unroll_scan* is True, backpropagation always includes the full sequence, so *gradient_steps* must be set to -1 and the input sequence length must be known at compile time (i.e., cannot be given as None).

precompute_input : bool

If True, precompute input_to_hid before iterating through the sequence. This can result in a speedup at the expense of an increase in memory usage.

mask_input : `lasagne.layers.Layer`

Layer which allows for a sequence mask to be input, for when sequences are of variable length. Default *None*, which means no mask will be supplied (i.e. all sequences are of the same length).

References

[R26]

class `lasagne.layers.`**`LSTMLayer`**(*incoming*, *num_units*, *ingate=lasagne.layers.Gate()*, *forgetgate=lasagne.layers.Gate()*, *cell=lasagne.layers.Gate(W_cell=None*, *nonlinearity=lasagne.nonlinearities.tanh)*, *outgate=lasagne.layers.Gate()*, *nonlinearity=lasagne.nonlinearities.tanh*, *cell_init=lasagne.init.Constant(0.)*, *hid_init=lasagne.init.Constant(0.)*, *backwards=False*, *learn_init=False*, *peepholes=True*, *gradient_steps=-1*, *grad_clipping=False*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, ***kwargs*)

A long short-term memory (LSTM) layer.

Includes optional "peephole connections" and a forget gate. Based on the definition in [R27], which is the current common definition. The output is computed by

$$
\begin{aligned}
i_t &= \sigma_i(x_t W_{xi} + h_{t-1} W_{hi} + w_{ci} \odot c_{t-1} + b_i) \\
f_t &= \sigma_f(x_t W_{xf} + h_{t-1} W_{hf} + w_{cf} \odot c_{t-1} + b_f) \\
c_t &= f_t \odot c_{t-1} + i_t \sigma_c(x_t W_{xc} + h_{t-1} W_{hc} + b_c) \\
o_t &= \sigma_o(x_t W_{xo} + h_{t-1} W_{ho} + w_{co} \odot c_t + b_o) \\
h_t &= o_t \odot \sigma_h(c_t)
\end{aligned}
$$

Parameters **incoming** : a `lasagne.layers.Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape.

num_units : int

Number of hidden/cell units in the layer.

ingate : Gate

Parameters for the input gate (i_t): W_{xi}, W_{hi}, w_{ci}, b_i, and σ_i.

forgetgate : Gate

Parameters for the forget gate (f_t): W_{xf}, W_{hf}, w_{cf}, b_f, and σ_f.

cell : Gate

Parameters for the cell computation (c_t): W_{xc}, W_{hc}, b_c, and σ_c.

outgate : Gate

Parameters for the output gate (o_t): W_{xo}, W_{ho}, w_{co}, b_o, and σ_o.

nonlinearity : callable or None

The nonlinearity that is applied to the output (σ_h). If None is provided, no nonlinearity will be applied.

cell_init : callable, np.ndarray, theano.shared or TensorVariable

Initializer for initial cell state (c_0). If a TensorVariable (Theano expression) is supplied, it will not be learned regardless of the value of *learn_init*.

hid_init : callable, np.ndarray, theano.shared or TensorVariable

Initializer for initial hidden state (h_0). If a TensorVariable (Theano expression) is supplied, it will not be learned regardless of the value of *learn_init*.

backwards : bool

If True, process the sequence backwards and then reverse the output again such that the output from the layer is always from x_1 to x_n.

learn_init : bool

If True, initial hidden values are learned. If *hid_init* or *cell_init* are TensorVariables then the TensorVariable is used and *learn_init* is ignored for that initial state.

peepholes : bool

If True, the LSTM uses peephole connections. When False, *ingate.W_cell*, *forgetgate.W_cell* and *outgate.W_cell* are ignored.

gradient_steps : int

Number of timesteps to include in the backpropagated gradient. If -1, backpropagate through the entire sequence.

grad_clipping: False or float

If a float is provided, the gradient messages are clipped during the backward pass. If False, the gradients will not be clipped. See [R27] (p. 6) for further explanation.

unroll_scan : bool

If True the recursion is unrolled instead of using scan. For some graphs this gives a significant speed up but it might also consume more memory. When *unroll_scan* is True, backpropagation always includes the full sequence, so *gradient_steps* must be set to -1 and the input sequence length must be known at compile time (i.e., cannot be given as None).

precompute_input : bool

If True, precompute input_to_hid before iterating through the sequence. This can result in a speedup at the expense of an increase in memory usage.

mask_input : `lasagne.layers.Layer`

Layer which allows for a sequence mask to be input, for when sequences are of variable length. Default *None*, which means no mask will be supplied (i.e. all sequences are of the same length).

References

[R27]

get_output_for(*inputs*, ***kwargs*)

Compute this layer's output function given a symbolic input variable

Parameters **inputs** : list of theano.TensorType

inputs[0] should always be the symbolic input variable. When this layer has a mask input (i.e. was instantiated with *mask_input != None*, indicating that the lengths of sequences in each batch vary), *inputs* should have length 2, where *inputs[1]* is the *mask*. The *mask* should be supplied as a Theano variable denoting whether each time step in each sequence in the batch is part of the sequence or not. *mask* should be a matrix of shape `(n_batch, n_time_steps)` where `mask[i, j] = 1` when `j <= (length of sequence i)` and `mask[i, j] = 0` when `j > (length of sequence i)`.

Returns **layer_output** : theano.TensorType

Symbolic output variable.

class `lasagne.layers.`**GRULayer**(*incoming*, *num_units*, *resetgate=lasagne.layers.Gate(W_cell=None)*, *updategate=lasagne.layers.Gate(W_cell=None)*, *hidden_update=lasagne.layers.Gate(W_cell=None, lasagne.nonlinearities.tanh)*, *hid_init=lasagne.init.Constant(0.)*, *backwards=False*, *learn_init=True*, *gradient_steps=-1*, *grad_clipping=False*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, ***kwargs*)

Gated Recurrent Unit (GRU) Layer

Implements the recurrent step proposed in [R28], which computes the output by

$$
\begin{aligned}
r_t &= \sigma_r(x_t W_{xr} + h_{t-1} W_{hr} + b_r) \\
u_t &= \sigma_u(x_t W_{xu} + h_{t-1} W_{hu} + b_u) \\
c_t &= \sigma_c(x_t W_{xc} + r_t \odot (h_{t-1} W_{hc}) + b_c) \\
h_t &= (1 - u_t) \odot h_{t-1} + u_t \odot c_t
\end{aligned}
$$

Parameters **incoming** : a `lasagne.layers.Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape.

num_units : int

Number of hidden units in the layer.

resetgate : Gate

Parameters for the reset gate (r_t): W_{xr}, W_{hr}, b_r, and σ_r.

updategate : Gate

Parameters for the update gate (u_t): W_{xu}, W_{hu}, b_u, and σ_u.

hidden_update : Gate

Parameters for the hidden update (c_t): W_{xc}, W_{hc}, b_c, and σ_c.

hid_init : callable, np.ndarray, theano.shared or TensorVariable

Initializer for initial hidden state (h_0). If a TensorVariable (Theano expression) is supplied, it will not be learned regardless of the value of *learn_init*.

backwards : bool

If True, process the sequence backwards and then reverse the output again such that the output from the layer is always from x_1 to x_n.

learn_init : bool

If True, initial hidden values are learned. If *hid_init* is a TensorVariable then the TensorVariable is used and *learn_init* is ignored.

gradient_steps : int

Number of timesteps to include in the backpropagated gradient. If -1, backpropagate through the entire sequence.

grad_clipping : False or float

If a float is provided, the gradient messages are clipped during the backward pass. If False, the gradients will not be clipped. See [R28] (p. 6) for further explanation.

unroll_scan : bool

If True the recursion is unrolled instead of using scan. For some graphs this gives a significant speed up but it might also consume more memory. When *unroll_scan* is True, backpropagation always includes the full sequence, so *gradient_steps* must be set to -1 and the input sequence length must be known at compile time (i.e., cannot be given as None).

precompute_input : bool

If True, precompute input_to_hid before iterating through the sequence. This can result in a speedup at the expense of an increase in memory usage.

mask_input : `lasagne.layers.Layer`

Layer which allows for a sequence mask to be input, for when sequences are of variable length. Default *None*, which means no mask will be supplied (i.e. all sequences are of the same length).

Notes

An alternate update for the candidate hidden state is proposed in [R29]:

$$c_t = \sigma_c(x_t W_{ic} + (r_t \odot h_{t-1})W_{hc} + b_c)$$

We use the formulation from [R28] because it allows us to do all matrix operations in a single dot product.

References

[R28], [R29], [R30]

get_output_for(*inputs*, ***kwargs*)

Compute this layer's output function given a symbolic input variable

Parameters **inputs** : list of theano.TensorType

inputs[0] should always be the symbolic input variable. When this layer has a mask input (i.e. was instantiated with *mask_input != None*, indicating that the lengths of sequences in each batch vary), *inputs* should have length 2, where *inputs[1]* is the *mask*. The *mask* should be supplied as a Theano variable denoting whether each time step in each sequence in the batch is part of the sequence or not. *mask* should be a matrix of shape `(n_batch, n_time_steps)` where `mask[i, j] = 1` when `j <= (length of sequence i)` and `mask[i, j] = 0` when `j > (length of sequence i)`.

Returns **layer_output** : theano.TensorType

Symbolic output variable.

class `lasagne.layers.`**`Gate`**(*W_in=lasagne.init.Normal(0.1)*, *W_hid=lasagne.init.Normal(0.1)*, *W_cell=lasagne.init.Normal(0.1)*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.sigmoid*)

Simple class to hold the parameters for a gate connection. We define a gate loosely as something which computes the linear mix of two inputs, optionally computes an element-wise product with a third, adds a bias, and applies a nonlinearity.

Parameters **W_in** : Theano shared variable, numpy array or callable

Initializer for input-to-gate weight matrix.

W_hid : Theano shared variable, numpy array or callable

Initializer for hidden-to-gate weight matrix.

W_cell : Theano shared variable, numpy array, callable, or None

Initializer for cell-to-gate weight vector. If None, no cell-to-gate weight vector will be stored.

b : Theano shared variable, numpy array or callable

Initializer for input gate bias vector.

nonlinearity : callable or None

The nonlinearity that is applied to the input gate activation. If None is provided, no nonlinearity will be applied.

References

[R31]

Examples

For `LSTMLayer` the bias of the forget gate is often initialized to a large positive value to encourage the layer initially remember the cell value, see e.g. [R31] page 15.

```
>>> import lasagne
>>> forget_gate = Gate(b=lasagne.init.Constant(5.0))
>>> l_lstm = LSTMLayer((10, 20, 30), num_units=10,
...                    forgetgate=forget_gate)
```

2.1.8 Noise layers

class lasagne.layers.**DropoutLayer**(*incoming*, *p=0.5*, *rescale=True*, ***kwargs*)
Dropout layer

Sets values to zero with probability p. See notes for disabling dropout during testing.

Parameters **incoming** : a Layer instance or a tuple

the layer feeding into this layer, or the expected input shape

p : float or scalar tensor

The probability of setting a value to zero

rescale : bool

If true the input is rescaled with input / (1-p) when deterministic is False.

Notes

The dropout layer is a regularizer that randomly sets input values to zero; see [R15], [R16] for why this might improve generalization. During training you should set deterministic to false and during testing you should set deterministic to true.

If rescale is true the input is scaled with input / (1-p) when deterministic is false, see references for further discussion. Note that this implementation scales the input at training time.

References

[R15], [R16]

get_output_for(*input*, *deterministic=False*, ***kwargs*)

Parameters **input** : tensor

output from the previous layer

deterministic : bool

If true dropout and scaling is disabled, see notes

lasagne.layers.**dropout**
alias of DropoutLayer

class lasagne.layers.**GaussianNoiseLayer**(*incoming*, *sigma=0.1*, ***kwargs*)
Gaussian noise layer.

Add zero-mean Gaussian noise of given standard deviation to the input [R17].

Parameters **incoming** : a Layer instance or a tuple

the layer feeding into this layer, or the expected input shape

sigma : float or tensor scalar

Standard deviation of added Gaussian noise

Notes

The Gaussian noise layer is a regularizer. During training you should set deterministic to false and during testing you should set deterministic to true.

References

[R17]

get_output_for(*input*, *deterministic=False*, ***kwargs*)

Parameters **input** : tensor

output from the previous layer

deterministic : bool

If true noise is disabled, see notes

2.1.9 Shape layers

class lasagne.layers.**ReshapeLayer**(*incoming*, *shape*, ***kwargs*)

A layer reshaping its input tensor to another tensor of the same total number of elements.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape

shape : tuple

The target shape specification. Each element can be one of:

- i, a positive integer directly giving the size of the dimension
- [i], a single-element list of int, denoting to use the size of the i th input dimension
- -1, denoting to infer the size for this dimension to match the total number of elements in the input tensor (cannot be used more than once in a specification)
- TensorVariable directly giving the size of the dimension

Notes

The tensor elements will be fetched and placed in C-like order. That is, reshaping *[1,2,3,4,5,6]* to shape *(2,3)* will result in a matrix *[[1,2,3],[4,5,6]]*, not in *[[1,3,5],[2,4,6]]* (Fortran-like order), regardless of the memory layout of the input tensor. For C-contiguous input, reshaping is cheap, for others it may require copying the data.

Examples

```
>>> from lasagne.layers import InputLayer, ReshapeLayer
>>> l_in = InputLayer((32, 100, 20))
>>> l1 = ReshapeLayer(l_in, ((32, 50, 40)))
>>> l1.output_shape
(32, 50, 40)
>>> l_in = InputLayer((None, 100, 20))
>>> l1 = ReshapeLayer(l_in, ([0], [1], 5, -1))
```

```
>>> l1.output_shape
(None, 100, 5, 4)
```

lasagne.layers.**reshape**
alias of ReshapeLayer

class lasagne.layers.**FlattenLayer**(*incoming*, *outdim=2*, ***kwargs*)
A layer that flattens its input. The leading `outdim-1` dimensions of the output will have the same shape as the input. The remaining dimensions are collapsed into the last dimension.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape.

outdim : int

The number of dimensions in the output.

See also:

flatten Shortcut

lasagne.layers.**flatten**
alias of FlattenLayer

class lasagne.layers.**DimshuffleLayer**(*incoming*, *pattern*, ***kwargs*)
A layer that rearranges the dimension of its input tensor, maintaining the same same total number of elements.

Parameters **incoming** : a Layer instance or a tuple

the layer feeding into this layer, or the expected input shape

pattern : tuple

The new dimension order, with each element giving the index of the dimension in the input tensor or *'x'* to broadcast it. For example *(3,2,1,0)* will reverse the order of a 4-dimensional tensor. Use *'x'* to broadcast, e.g. *(3,2,1,'x',0)* will take a 4 tensor of shape *(2,3,5,7)* as input and produce a tensor of shape *(7,5,3,1,2)* with the 4th dimension being broadcast-able. In general, all dimensions in the input tensor must be used to generate the output tensor. Omitting a dimension attempts to collapse it; this can only be done to broadcast-able dimensions, e.g. a 5-tensor of shape *(7,5,3,1,2)* with the 4th being broadcast-able can be shuffled with the pattern *(4,2,1,0)* collapsing the 4th dimension resulting in a tensor of shape *(2,3,5,7)*.

Examples

```
>>> from lasagne.layers import InputLayer, DimshuffleLayer
>>> l_in = InputLayer((2, 3, 5, 7))
>>> l1 = DimshuffleLayer(l_in, (3, 2, 1, 'x', 0))
>>> l1.output_shape
(7, 5, 3, 1, 2)
>>> l2 = DimshuffleLayer(l1, (4, 2, 1, 0))
>>> l2.output_shape
(2, 3, 5, 7)
```

lasagne.layers.**dimshuffle**
alias of DimshuffleLayer

class lasagne.layers.**PadLayer**(*incoming*, *width*, *val=0*, *batch_ndim=2*, ***kwargs*)

Pad all dimensions except the first `batch_ndim` with `width` zeros on both sides, or with another value specified in `val`. Individual padding for each dimension or edge can be specified using a tuple or list of tuples for `width`.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape

width : int, iterable of int, or iterable of tuple

Padding width. If an int, pads each axis symmetrically with the same amount in the beginning and end. If an iterable of int, defines the symmetric padding width separately for each axis. If an iterable of tuples of two ints, defines a seperate padding width for each beginning and end of each axis.

val : float

Value used for padding

batch_ndim : int

Dimensions up to this value are not padded. For padding convolutional layers this should be set to 2 so the sample and filter dimensions are not padded

lasagne.layers.**pad**

alias of `PadLayer`

class lasagne.layers.**SliceLayer**(*incoming*, *indices*, *axis=-1*, ***kwargs*)

Slices the input at a specific axis and at specific indices.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape

indices : int or slice instance

If an `int`, selects a single element from the given axis, dropping the axis. If a slice, selects all elements in the given range, keeping the axis.

axis : int

Specifies the axis from which the indices are selected.

Examples

```
>>> from lasagne.layers import SliceLayer, InputLayer
>>> l_in = InputLayer((2, 3, 4))
>>> SliceLayer(l_in, indices=0, axis=1).output_shape
... # equals input[:, 0]
(2, 4)
>>> SliceLayer(l_in, indices=slice(0, 1), axis=1).output_shape
... # equals input[:, 0:1]
(2, 1, 4)
>>> SliceLayer(l_in, indices=slice(-2, None), axis=-1).output_shape
... # equals input[..., -2:]
(2, 3, 2)
```

2.1.10 Merge layers

class lasagne.layers.**ConcatLayer**(*incomings*, *axis=1*, ***kwargs*)
Concatenates multiple inputs along the specified axis. Inputs should have the same shape except for the dimension specified in axis, which can have different sizes.

Parameters **incomings** : a list of Layer instances or tuples

The layers feeding into this layer, or expected input shapes

axis : int

Axis which inputs are joined over

lasagne.layers.**concat**
alias of ConcatLayer

class lasagne.layers.**ElemwiseMergeLayer**(*incomings*, *merge_function*, ***kwargs*)
This layer performs an elementwise merge of its input layers. It requires all input layers to have the same output shape.

Parameters **incomings** : a list of Layer instances or tuples

the layers feeding into this layer, or expected input shapes, with all incoming shapes being equal

merge_function : callable

the merge function to use. Should take two arguments and return the updated value. Some possible merge functions are theano.tensor: mul, add, maximum and minimum.

See also:

ElemwiseSumLayer Shortcut for sum layer.

class lasagne.layers.**ElemwiseSumLayer**(*incomings*, *coeffs=1*, ***kwargs*)
This layer performs an elementwise sum of its input layers. It requires all input layers to have the same output shape.

Parameters **incomings** : a list of Layer instances or tuples

the layers feeding into this layer, or expected input shapes, with all incoming shapes being equal

coeffs: list or scalar

A same-sized list of coefficients, or a single coefficient that is to be applied to all instances. By default, these will not be included in the learnable parameters of this layer.

Notes

Depending on your architecture, this can be used to avoid the more costly ConcatLayer. For example, instead of concatenating layers before a DenseLayer, insert separate DenseLayer instances of the same number of output units and add them up afterwards. (This avoids the copy operations in concatenation, but splits up the dot product.)

2.1.11 Embedding layers

class lasagne.layers.**EmbeddingLayer**(*incoming*, *input_size*, *output_size*, *W=lasagne.init.Normal()*, ***kwargs*)

A layer for word embeddings. The input should be an integer type Tensor variable.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape.

input_size: int

The Number of different embeddings. The last embedding will have index input_size - 1.

output_size : int

The size of each embedding.

W : Theano shared variable, numpy array or callable

The embedding matrix.

Examples

```
>>> from lasagne.layers import EmbeddingLayer, InputLayer, get_output
>>> import theano
>>> x = T.imatrix()
>>> l_in = InputLayer((3, ))
>>> W = np.arange(3*5).reshape((3, 5)).astype('float32')
>>> l1 = EmbeddingLayer(l_in, input_size=3, output_size=5, W=W)
>>> output = get_output(l1, x)
>>> f = theano.function([x], output)
>>> x_test = np.array([[0, 2], [1, 2]]).astype('int32')
>>> f(x_test)
array([[[  0.,   1.,   2.,   3.,   4.],
        [ 10.,  11.,  12.,  13.,  14.]],

       [[  5.,   6.,   7.,   8.,   9.],
        [ 10.,  11.,  12.,  13.,  14.]]], dtype=float32)
```

2.1.12 lasagne.layers.corrmm

class lasagne.layers.corrmm.**Conv2DMMLayer**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=False*, ***kwargs*)

2D convolutional layer

Performs a 2D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity. This is an alternative implementation which uses theano.sandbox.cuda.blas.GpuCorrMM directly.

Parameters **incoming** : a Layer instance or a tuple

The layer feeding into this layer, or the expected input shape. The output of this layer should be a 4D tensor, with shape (batch_size, num_input_channels, input_rows, input_columns).

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 2-element tuple specifying the size of the filters.

stride : int or iterable of int

An integer or a 2-element tuple specifying the stride of the convolution operation.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

A single integer results in symmetric zero-padding of the given size on all borders, a tuple of two integers allows different symmetric padding per dimension.

`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size on both sides (one less on the second side for an even filter size). When `stride=1`, this results in an output size equal to the input size.

`'valid'` is an alias for `0` (no padding / a valid convolution).

Note that `'full'` and `'same'` can be faster than equivalent integer values due to optimizations by Theano.

untie_biases : bool (default: False)

If `False`, the layer will have a bias parameter for each channel, which is shared across all positions in this channel. As a result, the *b* attribute will be a vector (1D).

If True, the layer will have separate bias parameters for each position in each channel. As a result, the *b* attribute will be a 3D tensor.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. This should initialize the layer weights to a 4D array with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If None is provided, the layer will have no biases. This should initialize the layer biases to a 1D array with shape `(num_filters,)` if *untied_biases* is set to `False`. If it is set to `True`, its shape should be `(num_filters, input_rows, input_columns)` instead. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

flip_filters : bool (default: False)

Whether to flip the filters and perform a convolution, or not to flip them and perform a correlation. Flipping adds a bit of overhead, so it is disabled by default. In most cases this does not make a difference anyway because the filters are learnt. However, `flip_filters` should be set to `True` if weights are loaded into it that were learnt using a regular `lasagne.layers.Conv2DLayer`, for example.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

Notes

Unlike `lasagne.layers.Conv2DLayer`, this layer properly supports `pad='same'`. It is not emulated. This should result in better performance.

Attributes

W	(Theano shared variable) Variable representing the filter weights.
b	(Theano shared variable) Variable representing the biases.

2.1.13 lasagne.layers.cuda_convnet

class `lasagne.layers.cuda_convnet.`**`Conv2DCCLayer`**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *pad=0*, *untie_biases=False*, *W=None*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *dimshuffle=True*, *flip_filters=False*, *partial_sum=1*, ***kwargs*)

2D convolutional layer

Performs a 2D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity. This is an alternative implementation which uses the cuda-convnet wrappers from pylearn2: `pylearn2.sandbox.cuda_convnet.filter_acts.FilterActs`.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape. This layer expects a 4D tensor as its input, with shape `(batch_size, num_input_channels, input_rows, input_columns)`. If automatic dimshuffling is disabled (see notes), the shape should be `(num_input_channels, input_rows, input_columns, batch_size)` instead (c01b axis order).

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 2-element tuple specifying the size of the filters. This layer does not support non-square filters.

stride : int or iterable of int

An integer or a 2-element tuple specifying the stride of the convolution operation. This layer does not support using different strides along both axes.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

A single integer results in symmetric zero-padding of the given size on all borders. This layer does not support using different amounts of padding along both axes, but for compatibility to other layers you can still specify the padding as a tuple of two same-valued integers.

`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size on both sides (one less on the second side for an even filter size). When `stride=1`, this results in an output size equal to the input size.

`'valid'` is an alias for `0` (no padding / a valid convolution).

Note that `'full'` and `'same'` can be faster than equivalent integer values due to optimizations by Theano.

untie_biases : bool (default: False)

If `False`, the layer will have a bias parameter for each channel, which is shared across all positions in this channel. As a result, the *b* attribute will be a vector (1D).

If `True`, the layer will have separate bias parameters for each position in each channel. As a result, the *b* attribute will be a 3D tensor.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. This should initialize the layer weights to a 4D array with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. If automatic dimshuffling is disabled (see notes), the shape should be `(num_input_channels, input_rows, input_columns, num_filters)` instead (c01b axis order). See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If None is provided, the layer will have no biases. This should initialize the layer biases to a 1D array with shape `(num_filters,)` if *untied_biases* is set to `False`. If it is set to `True`, its shape should be `(num_filters, input_rows, input_columns)` instead. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

dimshuffle : bool (default: True)

If `True`, the layer will automatically apply the necessary dimshuffle operations to deal with the fact that the cuda-convnet implementation uses c01b (batch-size-last) axis order instead of bc01 (batch-size-first), which is the Lasagne/Theano default. This makes the layer interoperable with other Lasagne layers.

If `False`, this automatic dimshuffling is disabled and the layer will expect its input and parameters to have c01b axis order. It is up to the user to ensure this.

ShuffleBC01ToC01BLayer and ShuffleC01BToBC01Layer can be used to convert between bc01 and c01b axis order.

flip_filters : bool (default: False)

Whether to flip the filters and perform a convolution, or not to flip them and perform a correlation. Flipping adds a bit of overhead, so it is disabled by default. In most cases this does not make a difference anyway because the filters are learnt. However, `flip_filters` should be set to `True` if weights are loaded into it that were learnt using a regular `lasagne.layers.Conv2DLayer`, for example.

partial_sum : int or None (default: 1)

This value tunes the trade-off between memory usage and performance. You can specify any positive integer that is a divisor of the output feature map size (i.e. output rows times output columns). Higher values decrease memory usage, but also performance. Specifying 0 or `None` means the highest possible value will be used. The Lasagne default of `1` gives the best performance, but also the highest memory usage.

More information about this parameter can be found in the cuda-convnet documentation.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

Notes

Unlike `lasagne.layers.Conv2DLayer`, this layer properly supports `pad='same'`. It is not emulated. This should result in better performance.

The cuda-convnet convolution implementation has several limitations:

- only square filters are supported.
- only identical strides in the horizontal and vertical direction are supported.
- the number of filters must be a multiple of 16.
- the number of input channels must be even, or less than or equal to 3.
- if the gradient w.r.t. the input is to be computed, the number of channels must be divisible by 4.
- performance is optimal when the batch size is a multiple of 128 (but other batch sizes are supported).
- this layer only works on the GPU.

The cuda-convnet convolution implementation uses c01b (batch-size-last) axis order by default. The Theano/Lasagne default is bc01 (batch-size-first). This layer automatically adds the necessary dimshuffle operations for the input and the parameters so that it is interoperable with other layers that assume bc01 axis order. However, these additional dimshuffle operations may sometimes negatively affect performance. For this reason, it is possible to disable them by setting `dimshuffle=False`. In this case, the user is expected to manually ensure that the input and parameters have the correct axis order. `ShuffleBC01ToC01BLayer` and `ShuffleC01BToBC01Layer` can be used to convert between bc01 and c01b axis order.

Attributes

W	(Theano shared variable) Variable representing the filter weights.
b	(Theano shared variable) Variable representing the biases.

class lasagne.layers.cuda_convnet.**MaxPool2DCCLayer**(*incoming*, *pool_size*, *stride=None*, *ignore_border=False*, *dimshuffle=True*, ***kwargs*)

2D max-pooling layer

Performs 2D max-pooling over the two trailing axes of a 4D input tensor (or over axis 1 and 2 if `dimshuffle=False`, see notes). This is an alternative implementation which uses the cuda-convnet wrappers from pylearn2: `pylearn2.sandbox.cuda_convnet.pool.MaxPool`.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer or iterable

The length of the pooling region in each dimension. If an integer, it is promoted to a square pooling region. If an iterable, it should have two elements. This layer does not support non-square pooling regions.

stride : integer, iterable or `None`

The strides between sucessive pooling regions in each dimension. If `None` then `stride = pool_size`. This layer does not support using different strides along both axes.

pad : integer or iterable (default: 0)

This implementation does not support custom padding, so this argument must always be set to `0`. It exists only to make sure the interface is compatible with `lasagne.layers.MaxPool2DLayer`.

ignore_border : bool (default: False)

This implementation always includes partial pooling regions, so this argument must always be set to False. It exists only to make sure the interface is compatible with `lasagne.layers.MaxPool2DLayer`.

dimshuffle : bool (default: True)

If `True`, the layer will automatically apply the necessary dimshuffle operations to deal with the fact that the cuda-convnet implementation uses c01b (batch-size-last) axis order instead of bc01 (batch-size-first), which is the Lasagne/Theano default. This makes the layer interoperable with other Lasagne layers.

If `False`, this automatic dimshuffling is disabled and the layer will expect its input and parameters to have c01b axis order. It is up to the user to ensure this. `ShuffleBC01ToC01BLayer` and `ShuffleC01BToBC01Layer` can be used to convert between bc01 and c01b axis order.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

Notes

The cuda-convnet max-pooling implementation has several limitations:

- only square pooling regions are supported.
- only identical strides in the horizontal and vertical direction are supported.
- only square inputs are supported. (This limitation does not exist for the convolution implementation.)

- partial pooling regions are always included (`ignore_border` is forced to `False`).
- custom padding is not supported (`pad` is forced to `0`).
- this layer only works on the GPU.

The cuda-convnet pooling implementation uses c01b (batch-size-last) axis order by default. The Theano/Lasagne default is bc01 (batch-size-first). This layer automatically adds the necessary dimshuffle operations for the input and the parameters so that it is interoperable with other layers that assume bc01 axis order. However, these additional dimshuffle operations may sometimes negatively affect performance. For this reason, it is possible to disable them by setting `dimshuffle=False`. In this case, the user is expected to manually ensure that the input and parameters have the correct axis order. `ShuffleBC01ToC01BLayer` and `ShuffleC01BToBC01Layer` can be used to convert between bc01 and c01b axis order.

class `lasagne.layers.cuda_convnet.`**`ShuffleBC01ToC01BLayer`**(*incoming*, *name=None*)
shuffle 4D input from bc01 (batch-size-first) order to c01b (batch-size-last) order.

This layer can be used for interoperability between c01b and bc01 layers. For example, `MaxPool2DCCLayer` and `Conv2DCCLayer` operate in c01b mode when they are created with `dimshuffle=False`.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

`lasagne.layers.cuda_convnet.`**`bc01_to_c01b`**
alias of `ShuffleBC01ToC01BLayer`

class `lasagne.layers.cuda_convnet.`**`ShuffleC01BToBC01Layer`**(*incoming*, *name=None*)
shuffle 4D input from c01b (batch-size-last) order to bc01 (batch-size-first) order.

This layer can be used for interoperability between c01b and bc01 layers. For example, `MaxPool2DCCLayer` and `Conv2DCCLayer` operate in c01b mode when they are created with `dimshuffle=False`.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

`lasagne.layers.cuda_convnet.`**`c01b_to_bc01`**
alias of `ShuffleC01BToBC01Layer`

class `lasagne.layers.cuda_convnet.`**`NINLayer_c01b`**(*incoming*, *num_units*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, ***kwargs*)
Network-in-network layer with c01b axis ordering.

This is a c01b version of `lasagne.layers.NINLayer`.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape

num_units : int

The number of units of the layer

untie_biases : bool

If `False`, the network has a single bias vector similar to a dense layer. If `True`, a separate bias vector is used for each spatial position.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. If a shared variable or a numpy array is provided the shape should be (num_units, num_input_channels). See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If a shared variable or a numpy array is provided the correct shape is determined by the untie_biases setting. If untie_biases is `False`, then the shape should be `(num_units,)`. If untie_biases is `True` then the shape should be `(num_units, rows, columns)`. If `None` is provided the layer will have no biases. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

2.1.14 `lasagne.layers.dnn`

class `lasagne.layers.dnn.`**`Pool2DDNNLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0)*, *ignore_border=True*, *mode='max'*, ***kwargs*)

2D pooling layer

Performs 2D mean- or max-pooling over the two trailing axes of a 4D input tensor. This is an alternative implementation which uses `theano.sandbox.cuda.dnn.dnn_pool` directly.

Parameters **incoming** : a `Layer` instance or tuple

The layer feeding into this layer, or the expected input shape.

pool_size : integer or iterable

The length of the pooling region in each dimension. If an integer, it is promoted to a square pooling region. If an iterable, it should have two elements.

stride : integer, iterable or `None`

The strides between sucessive pooling regions in each dimension. If `None` then `stride = pool_size`.

pad : integer or iterable

Number of elements to be added on each side of the input in each dimension. Each value must be less than the corresponding stride.

ignore_border : bool (default: True)

This implementation never includes partial pooling regions, so this argument must always be set to True. It exists only to make sure the interface is compatible with `lasagne.layers.MaxPool2DLayer`.

mode : string

Pooling mode, one of 'max', 'average_inc_pad' or 'average_exc_pad'. Defaults to 'max'.

****kwargs**

Any additional keyword arguments are passed to the `Layer` superclass.

Notes

The value used to pad the input is chosen to be less than the minimum of the input, so that the output of each pooling region always corresponds to some element in the unpadded input region.

This is a drop-in replacement for `lasagne.layers.MaxPool2DLayer`. Its interface is the same, except it does not support the `ignore_border` argument.

class `lasagne.layers.dnn.`**`MaxPool2DDNNLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0)*, *ignore_border=True*, ***kwargs*)

2D max-pooling layer

Subclass of `Pool2DDNNLayer` fixing `mode='max'`, provided for compatibility to other `MaxPool2DLayer` classes.

class `lasagne.layers.dnn.`**`Conv2DDNNLayer`**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=False*, ***kwargs*)

2D convolutional layer

Performs a 2D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity. This is an alternative implementation which uses `theano.sandbox.cuda.dnn.dnn_conv` directly.

Parameters **incoming** : a `Layer` instance or a tuple

The layer feeding into this layer, or the expected input shape. The output of this layer should be a 4D tensor, with shape `(batch_size, num_input_channels, input_rows, input_columns)`.

num_filters : int

The number of learnable convolutional filters this layer has.

filter_size : int or iterable of int

An integer or a 2-element tuple specifying the size of the filters.

stride : int or iterable of int

An integer or a 2-element tuple specifying the stride of the convolution operation.

pad : int, iterable of int, 'full', 'same' or 'valid' (default: 0)

By default, the convolution is only computed where the input and the filter fully overlap (a valid convolution). When `stride=1`, this yields an output that is smaller than the input by `filter_size - 1`. The *pad* argument allows you to implicitly pad the input with zeros, extending the output size.

A single integer results in symmetric zero-padding of the given size on all borders, a tuple of two integers allows different symmetric padding per dimension.

`'full'` pads with one less than the filter size on both sides. This is equivalent to computing the convolution wherever the input and the filter overlap by at least one position.

`'same'` pads with half the filter size on both sides (one less on the second side for an even filter size). When `stride=1`, this results in an output size equal to the input size.

`'valid'` is an alias for `0` (no padding / a valid convolution).

Note that `'full'` and `'same'` can be faster than equivalent integer values due to optimizations by Theano.

untie_biases : bool (default: False)

If `False`, the layer will have a bias parameter for each channel, which is shared across all positions in this channel. As a result, the *b* attribute will be a vector (1D).

If True, the layer will have separate bias parameters for each position in each channel. As a result, the *b* attribute will be a 3D tensor.

W : Theano shared variable, numpy array or callable

An initializer for the weights of the layer. This should initialize the layer weights to a 4D array with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b : Theano shared variable, numpy array, callable or None

An initializer for the biases of the layer. If None is provided, the layer will have no biases. This should initialize the layer biases to a 1D array with shape `(num_filters,)` if *untied_biases* is set to `False`. If it is set to `True`, its shape should be `(num_filters, input_rows, input_columns)` instead. See `lasagne.utils.create_param()` for more information.

nonlinearity : callable or None

The nonlinearity that is applied to the layer activations. If None is provided, the layer will be linear.

flip_filters : bool (default: False)

Whether to flip the filters and perform a convolution, or not to flip them and perform a correlation. Flipping adds a bit of overhead, so it is disabled by default. In most cases this does not make a difference anyway because the filters are learnt. However, `flip_filters` should be set to `True` if weights are loaded into it that were learnt using a regular `lasagne.layers.Conv2DLayer`, for example.

****kwargs**

Any additional keyword arguments are passed to the *Layer* superclass.

Notes

Unlike `lasagne.layers.Conv2DLayer`, this layer properly supports `pad='same'`. It is not emulated. This should result in better performance.

Attributes

W	(Theano shared variable) Variable representing the filter weights.
b	(Theano shared variable) Variable representing the biases.

Helper functions

get_output	Computes the output of the network at one or more given layers.
get_output_shape	Computes the output shape of the network at one or more given layers.
get_all_layers	This function gathers all layers below one or more given Layer instances, including the given layer(
get_all_params	This function gathers all parameters of all layers below one or more given Layer instances, includin
count_params	This function counts all parameters (i.e., the number of scalar values) of all layers below one or more
get_all_param_values	This function returns the values of the parameters of all layers below one or more given Layer insta
set_all_param_values	Given a list of numpy arrays, this function sets the parameters of all layers below one or more given L

Layer base classes

Layer	The Layer class represents a single layer of a neural network.
MergeLayer	This class represents a layer that aggregates input from multiple layers.

Network input

InputLayer	This layer holds a symbolic variable that represents a network input.

Dense layers

DenseLayer	A fully connected layer.
NonlinearityLayer	A layer that just applies a nonlinearity.
NINLayer	Network-in-network layer.

Convolutional layers

Conv1DLayer	1D convolutional layer
Conv2DLayer	2D convolutional layer

Pooling layers

MaxPool1DLayer	1D max-pooling layer
MaxPool2DLayer	2D max-pooling layer
Pool2DLayer	2D pooling layer
GlobalPoolLayer	Global pooling layer
FeaturePoolLayer	Feature pooling layer
FeatureWTALayer	'Winner Take All' layer

Recurrent layers

CustomRecurrentLayer	A layer which implements a recurrent connection.
RecurrentLayer	Dense recurrent neural network (RNN) layer

Continued on next page

Table 2.9 – continued from previous page

LSTMLayer	A long short-term memory (LSTM) layer.
GRULayer	Gated Recurrent Unit (GRU) Layer
Gate	Simple class to hold the parameters for a gate connection.

Noise layers

DropoutLayer	Dropout layer
dropout	alias of DropoutLayer
GaussianNoiseLayer	Gaussian noise layer.

Shape layers

ReshapeLayer	A layer reshaping its input tensor to another tensor of the same total number of elements.
reshape	alias of ReshapeLayer
FlattenLayer	A layer that flattens its input.
flatten	alias of FlattenLayer
DimshuffleLayer	A layer that rearranges the dimension of its input tensor, maintaining the same same total number of element
dimshuffle	alias of DimshuffleLayer
PadLayer	Pad all dimensions except the first batch_ndim with width zeros on both sides, or with another value spe
pad	alias of PadLayer
SliceLayer	Slices the input at a specific axis and at specific indices.

Merge layers

ConcatLayer	Concatenates multiple inputs along the specified axis.
concat	alias of ConcatLayer
ElemwiseMergeLayer	This layer performs an elementwise merge of its input layers.
ElemwiseSumLayer	This layer performs an elementwise sum of its input layers.

Embedding layers

EmbeddingLayer	A layer for word embeddings.

lasagne.layers.corrmm

corrmm.Conv2DMMLayer	2D convolutional layer

lasagne.layers.cuda_convnet

cuda_convnet.Conv2DCCLayer	2D convolutional layer
cuda_convnet.MaxPool2DCCLayer	2D max-pooling layer
cuda_convnet.ShuffleBC01ToC01BLayer	shuffle 4D input from bc01 (batch-size-first) order to c01b

Continued on next page

Table 2.15 – continued from previous page

cuda_convnet.bc01_to_c01b	alias of ShuffleBC01ToC01BLayer
cuda_convnet.ShuffleC01BToBC01Layer	shuffle 4D input from c01b (batch-size-last) order to bc01
cuda_convnet.c01b_to_bc01	alias of ShuffleC01BToBC01Layer
cuda_convnet.NINLayer_c01b	Network-in-network layer with c01b axis ordering.

lasagne.layers.dnn

dnn.Conv2DDNNLayer	2D convolutional layer
dnn.MaxPool2DDNNLayer	2D max-pooling layer
dnn.Pool2DDNNLayer	2D pooling layer

2.2 lasagne.updates

Functions to generate Theano update dictionaries for training.

The update functions implement different methods to control the learning rate for use with stochastic gradient descent.

Update functions take a loss expression or a list of gradient expressions and a list of parameters as input and return an ordered dictionary of updates:

sgd	Stochastic Gradient Descent (SGD) updates
momentum	Stochastic Gradient Descent (SGD) updates with momentum
nesterov_momentum	Stochastic Gradient Descent (SGD) updates with Nesterov momentum
adagrad	Adagrad updates
rmsprop	RMSProp updates
adadelta	Adadelta updates
adam	Adam updates

Two functions can be used to further modify the updates to include momentum:

apply_momentum	Returns a modified update dictionary including momentum
apply_nesterov_momentum	Returns a modified update dictionary including Nesterov momentum

Finally, we provide two helper functions to constrain the norm of tensors:

norm_constraint	Max weight norm constraints and gradient clipping
total_norm_constraint	Rescales a list of tensors based on their combined norm

norm_constraint() can be used to constrain the norm of parameters (as an alternative to weight decay), or for a form of gradient clipping. total_norm_constraint() constrain the total norm of a list of tensors. This is often used when training recurrent neural networks.

2.2.1 Examples

```
>>> import lasagne
>>> import theano.tensor as T
>>> import theano
>>> from lasagne.nonlinearities import softmax
```

```
>>> from lasagne.layers import InputLayer, DenseLayer, get_output
>>> from lasagne.updates import sgd, apply_momentum
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=3, nonlinearity=softmax)
>>> x = T.matrix('x')  # shp: num_batch x num_features
>>> y = T.ivector('y') # shp: num_batch
>>> l_out = get_output(l1, x)
>>> params = lasagne.layers.get_all_params(l1)
>>> loss = T.mean(T.nnet.categorical_crossentropy(l_out, y))
>>> updates_sgd = sgd(loss, params, learning_rate=0.0001)
>>> updates = apply_momentum(updates_sgd, params, momentum=0.9)
>>> train_function = theano.function([x, y], updates=updates)
```

2.2.2 Update functions

lasagne.updates.**sgd**(*loss_or_grads*, *params*, *learning_rate*)

Stochastic Gradient Descent (SGD) updates

Generates update expressions of the form:

- `param := param - learning_rate * gradient`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

Returns OrderedDict

A dictionary mapping each parameter to its update expression

lasagne.updates.**momentum**(*loss_or_grads*, *params*, *learning_rate*, *momentum=0.9*)

Stochastic Gradient Descent (SGD) updates with momentum

Generates update expressions of the form:

- `velocity := momentum * velocity - learning_rate * gradient`
- `param := param + velocity`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

See also:

`apply_momentum` Generic function applying momentum to updates

`nesterov_momentum` Nesterov's variant of SGD with momentum

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

`lasagne.updates.`**`nesterov_momentum`**(*loss_or_grads*, *params*, *learning_rate*, *momentum=0.9*)

Stochastic Gradient Descent (SGD) updates with Nesterov momentum

Generates update expressions of the form:

- `velocity := momentum * velocity + updates[param] - param`
- `param := param + momentum * velocity + updates[param] - param`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

See also:

`apply_nesterov_momentum` Function applying momentum to updates

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

The classic formulation of Nesterov momentum (or Nesterov accelerated gradient) requires the gradient to be evaluated at the predicted next position in parameter space. Here, we use the formulation described at https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617, which allows the gradient to be evaluated at the current parameters.

lasagne.updates.**adagrad**(*loss_or_grads*, *params*, *learning_rate=1.0*, *epsilon=1e-06*)

Adagrad updates

Scale learning rates by dividing with the square root of accumulated squared gradients. See [R42] for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

Using step size eta Adagrad calculates the learning rate for feature i at time step t as:

$$\eta_{t,i} = \frac{\eta}{\sqrt{\sum_{t'}^{t} g_{t',i}^2 + \epsilon}} g_{t,i}$$

as such the learning rate is monotonically decreasing.

Epsilon is not included in the typical formula, see [R43].

References

[R42], [R43]

lasagne.updates.**rmsprop**(*loss_or_grads*, *params*, *learning_rate=1.0*, *rho=0.9*, *epsilon=1e-06*)

RMSProp updates

Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients. See [R44] for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

rho : float or symbolic scalar

Gradient moving average decay factor

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

rho should be between 0 and 1. A value of *rho* close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

Using the step size η and a decay factor ρ the learning rate η_t is calculated as:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \frac{\eta}{\sqrt{r_t + \epsilon}}$$

References

[R44]

`lasagne.updates.`**`adadelta`**(*loss_or_grads*, *params*, *learning_rate=1.0*, *rho=0.95*, *epsilon=1e-06*)

Adadelta updates

Scale learning rates by a the ratio of accumulated gradients to accumulated step sizes, see [R45] and notes for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

rho : float or symbolic scalar

Squared gradient moving average decay factor

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

rho should be between 0 and 1. A value of rho close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

rho = 0.95 and epsilon=1e-6 are suggested in the paper and reported to work for multiple datasets (MNIST, speech).

In the paper, no learning rate is considered (so learning_rate=1.0). Probably best to keep it at this value. epsilon is important for the very first update (so the numerator does not become 0).

Using the step size eta and a decay factor rho the learning rate is calculated as:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \eta \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{r_t + \epsilon}}$$
$$s_t = \rho s_{t-1} + (1-\rho) * g^2$$

References

[R45]

`lasagne.updates.`**`adam`**(*loss_or_grads*, *params*, *learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)

Adam updates

Adam updates implemented as in [R46].

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float

Learning rate

beta_1 : float

Exponential decay rate for the first moment estimates.

beta_2 : float

Exponential decay rate for the second moment estimates.

epsilon : float

Constant for numerical stability.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

The paper [R46] includes an additional hyperparameter lambda. This is only needed to prove convergence of the algorithm and has no practical use (personal communication with the authors), it is therefore omitted here.

References

[R46]

2.2.3 Update modification functions

lasagne.updates.**apply_momentum**(*updates*, *params=None*, *momentum=0.9*)

Returns a modified update dictionary including momentum

Generates update expressions of the form:

- `velocity := momentum * velocity + updates[param] - param`
- `param := param + velocity`

Parameters **updates** : OrderedDict

A dictionary mapping parameters to update expressions

params : iterable of shared variables, optional

The variables to apply momentum to. If omitted, will apply momentum to all *updates.keys()*.

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A copy of *updates* with momentum updates for all *params*.

See also:

momentum Shortcut applying momentum to SGD updates

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

lasagne.updates.**apply_nesterov_momentum**(*updates*, *params=None*, *momentum=0.9*)

Returns a modified update dictionary including Nesterov momentum

Generates update expressions of the form:

- `velocity := momentum * velocity + updates[param] - param`
- `param := param + momentum * velocity + updates[param] - param`

Parameters **updates** : OrderedDict

A dictionary mapping parameters to update expressions

params : iterable of shared variables, optional

The variables to apply momentum to. If omitted, will apply momentum to all *updates.keys()*.

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A copy of *updates* with momentum updates for all *params*.

See also:

`nesterov_momentum` Shortcut applying Nesterov momentum to SGD updates

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

The classic formulation of Nesterov momentum (or Nesterov accelerated gradient) requires the gradient to be evaluated at the predicted next position in parameter space. Here, we use the formulation described at https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617, which allows the gradient to be evaluated at the current parameters.

2.2.4 Helper functions

`lasagne.updates.`**`norm_constraint`**(*tensor_var*, *max_norm*, *norm_axes=None*, *epsilon=1e-07*)

Max weight norm constraints and gradient clipping

This takes a TensorVariable and rescales it so that incoming weight norms are below a specified constraint value. Vectors violating the constraint are rescaled so that they are within the allowed range.

Parameters **tensor_var** : TensorVariable

Theano expression for update, gradient, or other quantity.

max_norm : scalar

This value sets the maximum allowed value of any norm in *tensor_var*.

norm_axes : sequence (list or tuple)

The axes over which to compute the norm. This overrides the default norm axes defined for the number of dimensions in *tensor_var*. When this is not specified and *tensor_var* is a matrix (2D), this is set to *(0,)*. If *tensor_var* is a 3D, 4D or 5D tensor, it is set to a tuple listing all axes but axis 0. The former default is useful for working with dense layers, the latter is useful for 1D, 2D and 3D convolutional layers. (Optional)

epsilon : scalar, optional

Value used to prevent numerical instability when dividing by very small or zero norms.

Returns TensorVariable

Input *tensor_var* with rescaling applied to weight vectors that violate the specified constraints.

Notes

When *norm_axes* is not specified, the axes over which the norm is computed depend on the dimensionality of the input variable. If it is 2D, it is assumed to come from a dense layer, and the norm is computed over axis 0. If it is 3D, 4D or 5D, it is assumed to come from a convolutional layer and the norm is computed over all trailing axes beyond axis 0. For other uses, you should explicitly specify the axes over which to compute the norm using *norm_axes*.

Examples

```
>>> param = theano.shared(
...     np.random.randn(100, 200).astype(theano.config.floatX))
>>> update = param + 100
>>> update = norm_constraint(update, 10)
>>> func = theano.function([], [], updates=[(param, update)])
>>> # Apply constrained update
>>> _ = func()
>>> from lasagne.utils import compute_norms
>>> norms = compute_norms(param.get_value())
>>> np.isclose(np.max(norms), 10)
True
```

lasagne.updates.**total_norm_constraint**(*tensor_vars*, *max_norm*, *epsilon=1e-07*, *return_norm=False*)

Rescales a list of tensors based on their combined norm

If the combined norm of the input tensors exceeds the threshold then all tensors are rescaled such that the combined norm is equal to the threshold.

Scaling the norms of the gradients is often used when training recurrent neural networks [R47].

Parameters **tensor_vars** : List of TensorVariables.

Tensors to be rescaled.

threshold : float

Threshold value for total norm.

epsilon : scalar, optional

Value used to prevent numerical instability when dividing by very small or zero norms.

return_norm : bool

If true the total norm is also returned.

Returns **tensor_vars_scaled** : list of TensorVariables

The scaled tensor variables.

norm : Theano scalar

The combined norms of the input variables prior to rescaling, only returned if `return_norms=True`.

Notes

The total norm can be used to monitor training.

References

[R47]

Examples

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> import lasagne
>>> from lasagne.updates import sgd, total_norm_constraint
>>> x = T.matrix()
>>> y = T.ivector()
>>> l_in = InputLayer((5, 10))
>>> l1 = DenseLayer(l_in, num_units=7, nonlinearity=T.nnet.softmax)
>>> output = lasagne.layers.get_output(l1, x)
>>> cost = T.mean(T.nnet.categorical_crossentropy(output, y))
>>> all_params = lasagne.layers.get_all_params(l1)
>>> all_grads = T.grad(cost, all_params)
>>> scaled_grads = total_norm_constraint(all_grads, 5)
>>> updates = sgd(scaled_grads, all_params, learning_rate=0.1)
```

2.3 lasagne.init

Functions to create initializers for parameter variables.

2.3.1 Examples

```
>>> from lasagne.layers import DenseLayer
>>> from lasagne.init import Constant, GlorotUniform
>>> l1 = DenseLayer((100,20), num_units=50, W=GlorotUniform(), b=Constant(0.0))
```

class lasagne.init.**Initializer**

Base class for parameter tensor initializers.

The Initializer class represents a weight initializer used to initialize weight parameters in a neural network layer. It should be subclassed when implementing new types of weight initializers.

sample(*shape*)

Sample should return a theano.tensor of size shape and data type theano.config.floatX.

Parameters **shape** : tuple or int

Integer or tuple specifying the size of the returned matrix.

returns : theano.tensor

Matrix of size shape and dtype theano.config.floatX.

class lasagne.init.**Constant**(*val=0.0*)

Initialize weights with constant value.

Parameters **val** : float

Constant value for weights.

class lasagne.init.**Normal**(*std=0.01*, *mean=0.0*)

Sample initial weights from the Gaussian distribution.

Initial weight parameters are sampled from N(mean, std).

Parameters **std** : float

Std of initial parameters.

mean : float

Mean of initial parameters.

class `lasagne.init.`**`Uniform`**(*range=0.01*, *std=None*, *mean=0.0*)
Sample initial weights from the uniform distribution.

Parameters are sampled from U(a, b).

Parameters **range** : float or tuple

When std is None then range determines a, b. If range is a float the weights are sampled from U(-range, range). If range is a tuple the weights are sampled from U(range[0], range[1]).

std : float or None

If std is a float then the weights are sampled from U(mean - np.sqrt(3) * std, mean + np.sqrt(3) * std).

mean : float

see std for description.

class `lasagne.init.`**`Glorot`**(*initializer*, *gain=1.0*, *c01b=False*)
Glorot weight initialization [R1].

This is also known as Xavier initialization.

Parameters **initializer** : lasagne.init.Initializer

Initializer used to sample the weights, must accept *std* in its constructor to sample from a distribution with a given standard deviation.

gain : float or 'relu'

Scaling factor for the weights. Set this to 1.0 for linear and sigmoid units, to 'relu' or sqrt(2) for rectified linear units. Other transfer functions may need different factors.

c01b : bool

For a `lasagne.layers.cuda_convnet.Conv2DCCLayer` constructed with `dimshuffle=False`, *c01b* must be set to `True` to compute the correct fan-in and fan-out.

See also:

`GlorotNormal` Shortcut with Gaussian initializer.

`GlorotUniform` Shortcut with uniform initializer.

Notes

For a `DenseLayer`, if `gain='relu'` and `initializer=Uniform`, the weights are initialized as

$$a = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$$
$$W \sim U[-a, a]$$

If `gain=1` and `initializer=Normal`, the weights are initialized as

$$\sigma = \sqrt{\frac{2}{fan_{in} + fan_{out}}}$$
$$W \sim N(0, \sigma)$$

References

[R1]

class `lasagne.init.`**GlorotNormal**(*gain=1.0*, *c01b=False*)
Glorot with weights sampled from the Normal distribution.

See `Glorot` for a description of the parameters.

class `lasagne.init.`**GlorotUniform**(*gain=1.0*, *c01b=False*)
Glorot with weights sampled from the Uniform distribution.

See `Glorot` for a description of the parameters.

class `lasagne.init.`**He**(*initializer*, *gain=1.0*, *c01b=False*)
He weight initialization [R2].

Weights are initialized with a standard deviation of $\sigma = gain\sqrt{\frac{1}{fan_{in}}}$.

Parameters **initializer** : lasagne.init.Initializer

Initializer used to sample the weights, must accept *std* in its constructor to sample from a distribution with a given standard deviation.

gain : float or 'relu'

Scaling factor for the weights. Set this to 1.0 for linear and sigmoid units, to 'relu' or sqrt(2) for rectified linear units. Other transfer functions may need different factors.

c01b : bool

For a `lasagne.layers.cuda_convnet.Conv2DCCLayer` constructed with `dimshuffle=False`, *c01b* must be set to `True` to compute the correct fan-in and fan-out.

See also:

HeNormal Shortcut with Gaussian initializer.

HeUniform Shortcut with uniform initializer.

References

[R2]

class `lasagne.init.`**HeNormal**(*gain=1.0*, *c01b=False*)
He initializer with weights sampled from the Normal distribution.

See `He` for a description of the parameters.

class `lasagne.init.`**HeUniform**(*gain=1.0*, *c01b=False*)
He initializer with weights sampled from the Uniform distribution.

See `He` for a description of the parameters.

class lasagne.init.**Orthogonal**(*gain=1.0*)
Intialize weights as Orthogonal matrix.

Orthogonal matrix initialization. For n-dimensional shapes where n > 2, the n-1 trailing axes are flattened. For convolutional layers, this corresponds to the fan-in, so this makes the initialization usable for both dense and convolutional layers.

Parameters **gain** : float or 'relu'

'relu' gives gain of sqrt(2).

class lasagne.init.**Sparse**(*sparsity=0.1*, *std=0.01*)
Initialize weights as sparse matrix.

Parameters **sparsity** : float

Exact fraction of non-zero values per column. Larger values give less sparsity.

std : float

Non-zero weights are sampled from N(0, std).

2.4 lasagne.nonlinearities

Non-linear activation functions for artificial neurons.

sigmoid(x)	Sigmoid activation function $\varphi(x) = \frac{1}{1+e^{-x}}$
softmax(x)	Softmax activation function $\varphi(\mathbf{x})_j = \frac{e^{\mathbf{x}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}_k}}$ where K is the total number of neurons in the layer.
tanh(x)	Tanh activation function $\varphi(x) = \tanh(x)$
rectify(x)	Rectify activation function $\varphi(x) = \max(0, x)$
LeakyRectify([leakiness])	Leaky rectifier $\varphi(x) = \max(\alpha \cdot x, x)$
leaky_rectify(x)	Instance of LeakyRectify with leakiness $\alpha = 0.01$
very_leaky_rectify(x)	Instance of LeakyRectify with leakiness $\alpha = 1/3$
linear(x)	Linear activation function $\varphi(x) = x$
identity(x)	Linear activation function $\varphi(x) = x$

2.4.1 Detailed description

lasagne.nonlinearities.**sigmoid**(*x*)
Sigmoid activation function $\varphi(x) = \frac{1}{1+e^{-x}}$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [0, 1]

The output of the sigmoid function applied to the activation.

lasagne.nonlinearities.**softmax**(*x*)
Softmax activation function $\varphi(\mathbf{x})_j = \frac{e^{\mathbf{x}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}_k}}$ where K is the total number of neurons in the layer. This activation function gets applied row-wise.

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 where the sum of the row is 1 and each single value is in [0, 1]

The output of the softmax function applied to the activation.

lasagne.nonlinearities.**tanh**(*x*)
Tanh activation function $\varphi(x) = \tanh(x)$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [-1, 1]

The output of the tanh function applied to the activation.

lasagne.nonlinearities.**rectify**(*x*)
Rectify activation function $\varphi(x) = \max(0, x)$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the rectify function applied to the activation.

class lasagne.nonlinearities.**LeakyRectify**(*leakiness=0.01*)
Leaky rectifier $\varphi(x) = \max(\alpha \cdot x, x)$

The leaky rectifier was introduced in [R34]. Compared to the standard rectifier `rectify()`, it has a nonzero gradient for negative input, which often helps convergence.

Parameters **leakiness** : float

Slope for negative input, usually between 0 and 1. A leakiness of 0 will lead to the standard rectifier, a leakiness of 1 will lead to a linear activation function, and any value in between will give a leaky rectifier.

See also:

leaky_rectify Instance with default leakiness of 0.01, as in [R34].

very_leaky_rectify Instance with high leakiness of 1/3, as in [R35].

References

[R34], [R35]

Examples

In contrast to other activation functions in this module, this is a class that needs to be instantiated to obtain a callable:

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((None, 100))
>>> from lasagne.nonlinearities import LeakyRectify
>>> custom_rectify = LeakyRectify(0.1)
>>> l1 = DenseLayer(l_in, num_units=200, nonlinearity=custom_rectify)
```

Alternatively, you can use the provided instance for leakiness=0.01:

```
>>> from lasagne.nonlinearities import leaky_rectify
>>> l2 = DenseLayer(l_in, num_units=200, nonlinearity=leaky_rectify)
```

Or the one for a high leakiness of 1/3:

```
>>> from lasagne.nonlinearities import very_leaky_rectify
>>> l3 = DenseLayer(l_in, num_units=200, nonlinearity=very_leaky_rectify)
```

Methods

__call__(x)	Apply the leaky rectify function to the activation x.

lasagne.nonlinearities.**leaky_rectify**(x)
Instance of LeakyRectify with leakiness $\alpha = 0.01$

lasagne.nonlinearities.**very_leaky_rectify**(x)
Instance of LeakyRectify with leakiness $\alpha = 1/3$

lasagne.nonlinearities.**linear**(x)
Linear activation function $\varphi(x) = x$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the identity applied to the activation.

2.5 lasagne.objectives

Provides some minimal help with building loss expressions for training or validating a neural network.

Three functions build element- or item-wise loss expressions from network predictions and targets:

binary_crossentropy	Computes the binary cross-entropy between predictions and targets.
categorical_crossentropy	Computes the categorical cross-entropy between predictions and targets.
squared_error	Computes the element-wise squared difference between two tensors.

A convenience function aggregates such losses into a scalar expression suitable for differentiation:

aggregate	Aggregates an element- or item-wise loss to a scalar loss.

Note that these functions only serve to write more readable code, but are completely optional. Essentially, any differentiable scalar Theano expression can be used as a training objective.

2.5.1 Examples

Assuming you have a simple neural network for 3-way classification:

```
>>> from lasagne.layers import InputLayer, DenseLayer, get_output
>>> from lasagne.nonlinearities import softmax, rectify
>>> l_in = InputLayer((100, 20))
>>> l_hid = DenseLayer(l_in, num_units=30, nonlinearity=rectify)
>>> l_out = DenseLayer(l_hid, num_units=3, nonlinearity=softmax)
```

And Theano variables representing your network input and targets:

```
>>> import theano
>>> data = theano.tensor.matrix('data')
>>> targets = theano.tensor.matrix('targets')
```

You'd first construct an element-wise loss expression:

```
>>> from lasagne.objectives import categorical_crossentropy, aggregate
>>> predictions = get_output(l_out, data)
>>> loss = categorical_crossentropy(predictions, targets)
```

Then aggregate it into a scalar (you could also just call `mean()` on it):

```
>>> loss = aggregate(loss, mode='mean')
```

Finally, this gives a loss expression you can pass to any of the update methods in `lasagne.updates`. For validation of a network, you will usually want to repeat these steps with deterministic network output, i.e., without dropout or any other nondeterministic computation in between:

```
>>> test_predictions = get_output(l_out, data, deterministic=True)
>>> test_loss = categorical_crossentropy(test_predictions, targets)
>>> test_loss = aggregate(test_loss)
```

This gives a loss expression good for monitoring validation error.

2.5.2 Loss functions

`lasagne.objectives.`**`binary_crossentropy`**(*predictions*, *targets*)

Computes the binary cross-entropy between predictions and targets.

$$L = -t \log(p) - (1 - t) \log(1 - p)$$

Parameters **predictions** : Theano tensor

Predictions in (0, 1), such as sigmoidal output of a neural network.

targets : Theano tensor

Targets in [0, 1], such as ground truth labels.

Returns Theano tensor

An expression for the element-wise binary cross-entropy.

Notes

This is the loss function of choice for binary classification problems and sigmoid output units.

`lasagne.objectives.`**`categorical_crossentropy`**(*predictions*, *targets*)

Computes the categorical cross-entropy between predictions and targets.

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$

Parameters **predictions** : Theano 2D tensor

Predictions in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets : Theano 2D tensor or 1D tensor

Either targets in [0, 1] matching the layout of *predictions*, or a vector of int giving the correct class index per data point.

Returns Theano 1D tensor

An expression for the item-wise categorical cross-entropy.

Notes

This is the loss function of choice for multi-class classification problems and softmax output units. For hard targets, i.e., targets that assign all of the probability to a single class per data point, providing a vector of int for the targets is usually slightly more efficient than providing a matrix with a single 1.0 per row.

`lasagne.objectives.`**`squared_error`**(*a*, *b*)

Computes the element-wise squared difference between two tensors.

$$L = (p - t)^2$$

Parameters **a, b** : Theano tensor

The tensors to compute the squared difference between.

Returns Theano tensor

An expression for the item-wise squared difference.

Notes

This is the loss function of choice for many regression problems or auto-encoders with linear output units.

2.5.3 Aggregation functions

`lasagne.objectives.`**`aggregate`**(*loss*, *weights=None*, *mode='mean'*)

Aggregates an element- or item-wise loss to a scalar loss.

Parameters **loss** : Theano tensor

The loss expression to aggregate.

weights : Theano tensor, optional

The weights for each element or item, must be broadcastable to the same shape as *loss* if given. If omitted, all elements will be weighted the same.

mode : {'mean', 'sum', 'normalized_sum'}

Whether to aggregate by averaging, by summing or by summing and dividing by the total weights (which requires *weights* to be given).

Returns Theano scalar

A scalar loss expression suitable for differentiation.

Notes

By supplying binary weights (i.e., only using values 0 and 1), this function can also be used for masking out particular entries in the loss expression. Note that masked entries still need to be valid values, not-a-numbers (NaNs) will propagate through.

When applied to batch-wise loss expressions, setting *mode* to `'normalized_sum'` ensures that the loss per batch is of a similar magnitude, independent of associated weights. However, it means that a given datapoint contributes more to the loss when it shares a batch with low-weighted or masked datapoints than with high-weighted ones.

2.6 lasagne.regularization

Functions to apply regularization to the weights in a network.

We provide functions to calculate the L1 and L2 penalty. Penalty functions take a tensor as input and calculate the penalty contribution from that tensor:

l1	Computes the L1 norm of a tensor
l2	Computes the squared L2 norm of a tensor

A helper function can be used to apply a penalty function to a tensor or a list of tensors:

apply_penalty	Computes the total cost for applying a specified penalty to a tensor or group of tensors.

Finally we provide two helper functions for applying a penalty function to the parameters in a layer or the parameters in a group of layers:

regularize_layer_params_weighted	Computes a regularization cost by applying a penalty to the parameters of a layer o
regularize_network_params	Computes a regularization cost by applying a penalty to the parameters of all layers

2.6.1 Examples

```
>>> import lasagne
>>> import theano.tensor as T
>>> import theano
>>> from lasagne.nonlinearities import softmax
>>> from lasagne.layers import InputLayer, DenseLayer, get_output
>>> from lasagne.regularization import regularize_layer_params_weighted, l2, l1
>>> from lasagne.regularization import regularize_layer_params
>>> layer_in = InputLayer((100, 20))
>>> layer1 = DenseLayer(layer_in, num_units=3)
>>> layer2 = DenseLayer(layer1, num_units=5, nonlinearity=softmax)
>>> x = T.matrix('x')  # shp: num_batch x num_features
>>> y = T.ivector('y') # shp: num_batch
>>> l_out = get_output(layer2, x)
>>> loss = T.mean(T.nnet.categorical_crossentropy(l_out, y))
>>> layers = {layer1: 0.1, layer2: 0.5}
>>> l2_penalty = regularize_layer_params_weighted(layers, l2)
>>> l1_penalty = regularize_layer_params(layer2, l1) * 1e-4
>>> loss = loss + l2_penalty + l1_penalty
```

2.6.2 Helper functions

lasagne.regularization.**apply_penalty**(*tensor_or_tensors*, *penalty*, ***kwargs*)

Computes the total cost for applying a specified penalty to a tensor or group of tensors.

Parameters **tensor_or_tensors** : Theano tensor or list of tensors

penalty : callable

****kwargs**

keyword arguments passed to penalty.

Returns Theano scalar

a scalar expression for the total penalty cost

lasagne.regularization.**regularize_layer_params**(*layer*, *penalty*, *tags={'regularizable': True}*, ***kwargs*)

Computes a regularization cost by applying a penalty to the parameters of a layer or group of layers.

Parameters **layer** : a `Layer` instances or list of layers.

penalty : callable

tags: dict

Tag specifications which filter the parameters of the layer or layers. By default, only parameters with the *regularizable* tag are included.

****kwargs**

keyword arguments passed to penalty.

Returns Theano scalar

a scalar expression for the cost

lasagne.regularization.**regularize_layer_params_weighted**(*layers*, *penalty*, *tags={'regularizable': True}*, ***kwargs*)

Computes a regularization cost by applying a penalty to the parameters of a layer or group of layers, weighted by a coefficient for each layer.

Parameters **layers** : dict

A mapping from `Layer` instances to coefficients.

penalty : callable

tags: dict

Tag specifications which filter the parameters of the layer or layers. By default, only parameters with the *regularizable* tag are included.

****kwargs**

keyword arguments passed to penalty.

Returns Theano scalar

a scalar expression for the cost

lasagne.regularization.**regularize_network_params**(*layer*, *penalty*, *tags={'regularizable': True}*, ***kwargs*)

Computes a regularization cost by applying a penalty to the parameters of all layers in a network.

Parameters **layer** : a `Layer` instance.

Parameters of this layer and all layers below it will be penalized.

penalty : callable

tags: dict

Tag specifications which filter the parameters of the layer or layers. By default, only parameters with the *regularizable* tag are included.

****kwargs**

keyword arguments passed to penalty.

Returns Theano scalar

a scalar expression for the cost

2.6.3 Penalty functions

`lasagne.regularization.l1(x)`
Computes the L1 norm of a tensor

Parameters **x** : Theano tensor

Returns Theano scalar

l1 norm (sum of absolute values of elements)

`lasagne.regularization.l2(x)`
Computes the squared L2 norm of a tensor

Parameters **x** : Theano tensor

Returns Theano scalar

squared l2 norm (sum of squared values of elements)

2.7 lasagne.random

A module with a package-wide random number generator, used for weight initialization and seeding noise layers. This can be replaced by a `numpy.random.RandomState` instance with a particular seed to facilitate reproducibility.

`lasagne.random.get_rng()`
Get the package-level random number generator.

Returns `numpy.random.RandomState` instance

The `numpy.random.RandomState` instance passed to the most recent call of `set_rng()`, or `numpy.random` if `set_rng()` has never been called.

`lasagne.random.set_rng(new_rng)`
Set the package-level random number generator.

Parameters **new_rng** : `numpy.random` or a `numpy.random.RandomState` instance

The random number generator to use.

2.8 lasagne.utils

lasagne.utils.**floatX**(*arr*)
Converts data to a numpy array of dtype `theano.config.floatX`.

Parameters **arr** : array_like

The data to be converted.

Returns numpy ndarray

The input array in the `floatX` dtype configured for Theano. If *arr* is an ndarray of correct dtype, it is returned as is.

lasagne.utils.**shared_empty**(*dim=2*, *dtype=None*)
Creates empty Theano shared variable.

Shortcut to create an empty Theano shared variable with the specified number of dimensions.

Parameters **dim** : int, optional

The number of dimensions for the empty variable, defaults to 2.

dtype : a numpy data-type, optional

The desired dtype for the variable. Defaults to the Theano `floatX` dtype.

Returns Theano shared variable

An empty Theano shared variable of dtype `dtype` with *dim* dimensions.

lasagne.utils.**as_theano_expression**(*input*)
Wrap as Theano expression.

Wraps the given input as a Theano constant if it is not a valid Theano expression already. Useful to transparently handle numpy arrays and Python scalars, for example.

Parameters **input** : number, numpy array or Theano expression

Expression to be converted to a Theano constant.

Returns Theano symbolic constant

Theano constant version of *input*.

lasagne.utils.**one_hot**(*x*, *m=None*)
One-hot representation of integer vector.

Given a vector of integers from 0 to m-1, returns a matrix with a one-hot representation, where each row corresponds to an element of x.

Parameters **x** : integer vector

The integer vector to convert to a one-hot representation.

m : int, optional

The number of different columns for the one-hot representation. This needs to be strictly greater than the maximum value of *x*. Defaults to `max(x) + 1`.

Returns Theano tensor variable

A Theano tensor variable of shape (`n`, *m*), where `n` is the length of *x*, with the one-hot representation of *x*.

Notes

If your integer vector represents target class memberships, and you wish to compute the cross-entropy between predictions and the target class memberships, then there is no need to use this function, since the function `lasagne.objectives.categorical_crossentropy()` can compute the cross-entropy from the integer vector directly.

`lasagne.utils.`**`unique`**(*l*)
Filters duplicates of iterable.

Create a new list from l with duplicate entries removed, while preserving the original order.

Parameters **l** : iterable

Input iterable to filter of duplicates.

Returns list

A list of elements of *l* without duplicates and in the same order.

`lasagne.utils.`**`compute_norms`**(*array*, *norm_axes=None*)
Compute incoming weight vector norms.

Parameters **array** : ndarray

Weight array.

norm_axes : sequence (list or tuple)

The axes over which to compute the norm. This overrides the default norm axes defined for the number of dimensions in *array*. When this is not specified and *array* is a 2D array, this is set to *(0,)*. If *array* is a 3D, 4D or 5D array, it is set to a tuple listing all axes but axis 0. The former default is useful for working with dense layers, the latter is useful for 1D, 2D and 3D convolutional layers. (Optional)

Returns **norms** : 1D array

1D array of incoming weight vector norms.

Examples

```
>>> array = np.random.randn(100, 200)
>>> norms = compute_norms(array)
>>> norms.shape
(200,)

>>> norms = compute_norms(array, norm_axes=(1,))
>>> norms.shape
(100,)
```

`lasagne.utils.`**`create_param`**(*spec*, *shape*, *name=None*)
Helper method to create Theano shared variables for layer parameters and to initialize them.

Parameters **spec** : numpy array, Theano shared variable, or callable

Either of the following:

- a numpy array with the initial parameter values
- a Theano shared variable representing the parameters

- a function or callable that takes the desired shape of the parameter array as its single argument and returns a numpy array.

shape : iterable of int

a tuple or other iterable of integers representing the desired shape of the parameter array.

name : string, optional

If a new variable is created, the name to give to the parameter variable. This is ignored if *spec* is already a Theano shared variable.

Returns Theano shared variable

a Theano shared variable representing layer parameters. If a numpy array was provided, the variable is initialized to contain this array. If a shared variable was provided, it is simply returned. If a callable was provided, it is called, and its output is used to initialize the variable.

Notes

This function is called by `Layer.add_param()` in the constructor of most `Layer` subclasses. This enables those layers to support initialization with numpy arrays, existing Theano shared variables, and callables for generating initial parameter values.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[Hinton2012] Improving neural networks by preventing co-adaptation of feature detectors. http://arxiv.org/abs/1207.0580

[R14] Lin, Min, Qiang Chen, and Shuicheng Yan (2013): Network in network. arXiv preprint arXiv:1312.4400.

[R25] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[R26] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[R27] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[R28] Cho, Kyunghyun, et al: On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259 (2014).

[R29] Chung, Junyoung, et al.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555 (2014).

[R30] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[R31] Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." Neural computation 12.10 (2000): 2451-2471.

[R15] Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. R. (2012): Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

[R16] Srivastava Nitish, Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2014): Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 5(Jun)(2), 1929-1958.

[R17] K.-C. Jim, C. Giles, and B. Horne (1996): An analysis of noise in recurrent neural networks: convergence and generalization. IEEE Transactions on Neural Networks, 7(6):1424-1438.

[R42] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12:2121-2159.

[R43] Chris Dyer: Notes on AdaGrad. http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf

[R44] Tieleman, T. and Hinton, G. (2012): Neural Networks for Machine Learning, Lecture 6.5 - rmsprop. Coursera. http://www.youtube.com/watch?v=O3sxAc4hxZU (formula @5:20)

[R45] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.

[R46] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[R47] Sutskever, I., Vinyals, O., & Le, Q. V. (2014): Sequence to sequence learning with neural networks. In Advances in Neural Information Processing Systems (pp. 3104-3112).

[R1] Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

[R2] Kaiming He et al. (2015): Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. arXiv preprint arXiv:1502.01852.

[R34] Maas et al. (2013): Rectifier Nonlinearities Improve Neural Network Acoustic Models, http://web.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf

[R35] Graham, Benjamin (2014): Spatially-sparse convolutional neural networks, http://arxiv.org/abs/1409.6070

l

H

I

L

M

N

O

P

R

S

T

U

V