lasagne Documentation

Release 0.2.dev1

Lasagne contributors

June 10, 2018

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.

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, but are otherwise very generic. For detailed step-by-step instructions for specific platforms including Windows, check our From Zero to Lasagne guides.

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.

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.

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`.

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.

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*!

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 "from theano.sandbox.cuda.dnn import dnn_available as d; print(d() or d.msg)"
```

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.

Docker

Instead of manually installing Theano and Lasagne on your machines as described above, you may want to use a pre-made Docker image: Lasagne Docker (CPU) or Lasagne Docker (CUDA). These are updated on a weekly basis with bleeding-edge builds of Theano and Lasagne. Examples of running bash in a Docker container are as follows:

```
sudo docker run -it kaixhin/lasagne
sudo nvidia-docker run -it kaixhin/cuda-lasagne:7.0
```

For a guide to Docker, see the official docs. CUDA support requires NVIDIA Docker. For more details on how to use the Lasagne Docker images, consult the source project.

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.

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) "More Examples" should get you covered! Graph Structures is a good extra read if you're curious about its inner workings.

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 train-images-idx3-ubyte.gz
Downloading train-labels-idx1-ubyte.gz
Downloading t10k-images-idx3-ubyte.gz
Downloading t10k-labels-idx1-ubyte.gz
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.

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.softmax)

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.

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.

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.

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*.

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.

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")
```

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)
```

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).

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.

Creating custom layers

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.

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.

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.

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)
```

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.

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.

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.

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.

lasagne.layers

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*, *unwrap_shared=True*, ***tags*)

Returns a list of Theano shared variables or expressions that parameterize the layer.

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 shared variables 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.

unwrap_shared [bool (default: True)] Affects only parameters that were set to a Theano expression. If `True` the function returns the shared variables contained in the expression, otherwise the Theano expression itself.

****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 or expressions representing the parameters.

Notes

If any of the layers' parameters was set to a Theano expression instead of a shared variable, *unwrap_shared* controls whether to return the shared variables involved in that expression (`unwrap_shared=True`, the default), or the expression itself (`unwrap_shared=False`). In either case, tag filtering applies to the expressions, considering all variables within an expression to be tagged the same.

Examples

Collecting all parameters from a two-layer network:

```
>>> from lasagne.layers import InputLayer, DenseLayer
>>> l_in = InputLayer((100, 20))
>>> l1 = DenseLayer(l_in, num_units=50)
>>> l2 = DenseLayer(l1, num_units=30)
>>> all_params = get_all_params(l2)
>>> all_params == [l1.W, l1.b, l2.W, l2.b]
True
```

Parameters can be filtered by tags, and parameter expressions are unwrapped to return involved shared variables by default:

```
>>> from lasagne.utils import floatX
>>> w1 = theano.shared(floatX(.01 * np.random.randn(50, 30)))
>>> w2 = theano.shared(floatX(1))
>>> l2 = DenseLayer(l1, num_units=30, W=theano.tensor.exp(w1) - w2, b=None)
>>> all_params = get_all_params(l2, regularizable=True)
>>> all_params == [l1.W, w1, w2]
True
```

When disabling unwrapping, the expression for `l2.W` is returned instead:

```
>>> all_params = get_all_params(l2, regularizable=True,
...                             unwrap_shared=False)
>>> all_params == [l1.W, l2.W]
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.
```

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.

Parameters

incoming [a Layer instance or a tuple] The layer feeding into this layer, or the expected input shape.

name [a string or None] An optional name to attach to this layer.

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

Register and possibly initialize a parameter tensor for the layer.

When defining a layer class, this method is called 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. This allows layer classes to transparently support parameter initialization from numpy arrays and callables, as well as setting parameters to existing Theano shared variables or Theano expressions.

All registered parameters are stored along with their tags in the ordered dictionary `Layer.params`, and can be retrieved with `Layer.get_params()`, optionally filtered by their tags.

Parameters

spec [Theano shared variable, expression, numpy array or callable] initial value, expression or initializer for this parameter. See `lasagne.utils.create_param()` for more information.

shape [tuple of int] a tuple of integers representing the desired shape of the parameter tensor.

name [str (optional)] a descriptive name for the parameter variable. This will be passed to `theano.shared` when the variable is created, prefixed by the layer's name if any (in the form `'layer_name.param_name'`). If `spec` is already a shared variable or expression, this parameter will be ignored to avoid overwriting an existing name.

****tags (optional)** tags associated with the parameter can be specified as keyword arguments. To associate the tag `tag1` with the parameter, pass `tag1=True`.

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

Returns

Theano shared variable or Theano expression the resulting parameter variable or parameter expression

Notes

It is recommended to assign the resulting parameter variable/expression to an attribute of the layer for easy access, 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(*unwrap_shared=True*, ***tags*)

Returns a list of Theano shared variables or expressions that parameterize the layer.

By default, all shared variables 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).

If any of the layer's parameters was set to a Theano expression instead of a shared variable, *unwrap_shared* controls whether to return the shared variables involved in that expression (`unwrap_shared=True`, the default), or the expression itself (`unwrap_shared=False`). In either case, tag filtering applies to the expressions, considering all variables within an expression to be tagged the same.

Parameters

unwrap_shared [bool (default: True)] Affects only parameters that were set to a Theano expression. If `True` the function returns the shared variables contained in the expression, otherwise the Theano expression itself.

****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 or expressions 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.

Parameters

incomings [a list of `Layer` instances or tuples] The layers feeding into this layer, or expected input shapes.

name [a string or None] An optional name to attach to this layer.

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*.

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))
```

Dense layers

class lasagne.layers.**DenseLayer**(*incoming*, *num_units*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *num_leading_axes=1*, ***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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a matrix with shape `(num_inputs, num_units)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be a 1D array with shape `(num_units,)`. 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.

num_leading_axes [int] Number of leading axes to distribute the dot product over. These axes will be kept in the output tensor, remaining axes will be collapsed and multiplied against the weight matrix. A negative number gives the (negated) number of trailing axes to involve in the dot product.

Examples

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

If the input has more than two axes, by default, all trailing axes will be flattened. This is useful when a dense layer follows a convolutional layer.

```
>>> l_in = InputLayer((None, 10, 20, 30))
>>> DenseLayer(l_in, num_units=50).output_shape
(None, 50)
```

Using the *num_leading_axes* argument, you can specify to keep more than just the first axis. E.g., to apply the same dot product to each step of a batch of time sequences, you would want to keep the first two axes.

```
>>> DenseLayer(l_in, num_units=50, num_leading_axes=2).output_shape
(None, 10, 50)
>>> DenseLayer(l_in, num_units=50, num_leading_axes=-1).output_shape
(None, 10, 20, 50)
```

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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a matrix with shape `(num_inputs, num_units)`, where `num_inputs` is the size of the second dimension of the input. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be a 1D array with shape `(num_units,)` for `untie_biases=False`, and a tensor of shape `(num_units, input_shape[2], ..., input_shape[-1])` for `untie_biases=True`. 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

[1]

Examples

```
>>> from lasagne.layers import InputLayer, NINLayer
>>> l_in = InputLayer((100, 20, 10, 3))
>>> l1 = NINLayer(l_in, num_units=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*, *flip_filters=True*, *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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 3D tensor with shape `(num_filters, num_input_channels, filter_length)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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.

flip_filters [bool (default: True)] Whether to flip the filters before sliding them over the input, performing a convolution (this is the default), or not to flip them and perform a correlation. Note that for some other convolutional layers in Lasagne, flipping incurs an overhead and is disabled by default – check the documentation when using learned weights from another layer.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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. Note that not all implementations support all settings for *pad*, *subsample* and *num_groups*.

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

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

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*, *flip_filters=True*, *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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 4D tensor with shape `(num_filters,`

num_input_channels, filter_rows, filter_columns). See lasagne.utils.create_param() for more information.

b [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for the biases. If set to None, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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: True)] Whether to flip the filters before sliding them over the input, performing a convolution (this is the default), or not to flip them and perform a correlation. Note that for some other convolutional layers in Lasagne, flipping incurs an overhead and is disabled by default – check the documentation when using learned weights from another layer.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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.

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

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.

---

class lasagne.layers.**Conv3DLayer**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1, 1)*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=True*, *convolution=theano.tensor.nnet.conv3d*, ***kwargs*)

3D convolutional layer

Performs a 3D 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 5D tensor, with shape (batch_size, num_input_channels, input_depth, 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 3-element tuple specifying the size of the filters.

stride [int or iterable of int] An integer or a 3-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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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 4D tensor.

W [Theano shared variable, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 5D tensor with shape `(num_filters, num_input_channels, filter_depth, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_depth, output_rows, output_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: True)] Whether to flip the filters before sliding them over the input, performing a convolution (this is the default), or not to flip them and perform a correlation. Note that for some other convolutional layers in Lasagne, flipping incurs an overhead and is disabled by default – check the documentation when using learned weights from another layer.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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.

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

class lasagne.layers.**TransposedConv2DLayer**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *crop=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=False*, ***kwargs*)

2D transposed convolution layer

Performs the backward pass of a 2D convolution (also called transposed convolution, fractionally-strided convolution or deconvolution in the literature) 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 transposed convolution operation. For the transposed convolution, this gives the dilation factor for the input – increasing it increases the output size.

crop [int, iterable of int, 'full', 'same' or 'valid' (default: 0)] By default, the transposed convolution is computed where the input and the filter overlap by at least one position (a full convolution). When `stride=1`, this yields an output that is larger than the input by `filter_size - 1`. It can be thought of as a valid convolution padded with zeros. The *crop* argument allows you to decrease the amount of this zero-padding, reducing the output size. It is the counterpart to the *pad* argument in a non-transposed convolution.

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

`'full'` disables zero-padding. It is is equivalent to computing the convolution wherever the input and the filter fully overlap.

`'same'` pads with half the filter size (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'valid'` is an alias for `0` (no cropping / a full 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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 4D tensor

with shape `(num_input_channels, num_filters, filter_rows, filter_columns)`. Note that the first two dimensions are swapped compared to a non-transposed convolution. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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 before sliding them over the input, performing a convolution, or not to flip them and perform a correlation (this is the default). Note that this flag is inverted compared to a non-transposed convolution.

output_size [int or iterable of int or symbolic tuple of ints] The output size of the transposed convolution. Allows to specify which of the possible output shapes to return when stride > 1. If not specified, the smallest shape will be returned.

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

Notes

The transposed convolution is implemented as the backward pass of a corresponding non-transposed convolution. It can be thought of as dilating the input (by adding `stride - 1` zeros between adjacent input elements), padding it with `filter_size - 1 - crop` zeros, and cross-correlating it with the filters. See [1] for more background.

References

[1]

Examples

To transpose an existing convolution, with tied filter weights:

```
>>> from lasagne.layers import Conv2DLayer, TransposedConv2DLayer
>>> conv = Conv2DLayer((None, 1, 32, 32), 16, 3, stride=2, pad=2)
>>> deconv = TransposedConv2DLayer(conv, conv.input_shape[1],
...         conv.filter_size, stride=conv.stride, crop=conv.pad,
...         W=conv.W, flip_filters=not conv.flip_filters)
```

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

`lasagne.layers.`**`Deconv2DLayer`**
alias of `TransposedConv2DLayer`

class lasagne.layers.**DilatedConv2DLayer**(*incoming*, *num_filters*, *filter_size*, *dilation=(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 dilated convolution layer

Performs a 2D convolution with dilated filters, then 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.

dilation [int or iterable of int] An integer or a 2-element tuple specifying the dilation factor of the filters. A factor of x corresponds to $x-1$ zeros inserted between adjacent filter elements.

pad [int, iterable of int, or 'valid' (default: 0)] The amount of implicit zero padding of the input. This implementation does not support padding, the argument is provided for compatibility to other convolutional layers only.

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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 4D tensor with shape `(num_input_channels, num_filters, filter_rows, filter_columns)`. Note that the first two dimensions are swapped compared to a non-dilated convolution. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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 before sliding them over the input, performing a convolution, or not to flip them and perform a correlation (this is the default). This implementation does not support flipped filters, the argument is provided for compatibility to other convolutional layers only.

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

Notes

The dilated convolution is implemented as the backward pass of a convolution wrt. weights, passing the filters as the output gradient. It can be thought of as dilating the filters (by adding `dilation - 1` zeros between adjacent filter elements) and cross-correlating them with the input. See [1] for more background.

References

[1]

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

Local layers

class `lasagne.layers.`**`LocallyConnected2DLayer`**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1)*, *pad='same'*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=True*, *channelwise=False*, ***kwargs*)

2D locally connected layer

Performs an operation similar to a 2D convolution but without the weight sharing, then 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. This implementation only supports unit stride, the argument is provided for compatibility to convolutional layers only.

pad [int, iterable of int, or 'valid' (default: 'same')] The amount of implicit zero padding of the input. This implementation only supports 'same' padding, the argument is provided for compatibility to other convolutional layers only.

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, expression, numpy array or callable] Initial value, expression or initializer for the weights. If `channelwise` is set to `False`, the weights should be a 6D tensor with shape `(num_filters, num_input_channels, filter_rows,`

`filter_columns, output_rows, output_columns)`. If `channelwise` is set to `True`, the weights should be a 5D tensor with shape `(num_filters, filter_rows, filter_columns, output_rows, output_columns)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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: True)] Whether to flip the filters before multiplying them over the input, similar to a convolution (this is the default), or not to flip them, similar to a correlation.

channelwise [bool (default: False)] If `False`, each filter interacts will all of the input channels as in a convolution. If `True`, each filter only interacts with the corresponding input channel. That is, each output channel only depends on its filter and on the input channel at the same channel index. In this case, the number of output channels (i.e. number of filters) should be equal to the number of input channels.

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

Raises

ValueError When `channelwise` is set to `True` and the number of filters differs from the number of input channels, a *ValueError* is raised.

Notes

This implementation computes the output tensor by iterating over the filter weights and multiplying them with shifted versions of the input tensor. This implementation assumes no stride, 'same' padding and no dilation.

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

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.`**`MaxPool3DLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0, 0)*, *ignore_border=True*, ***kwargs*)

3D max-pooling layer

Performs 3D max-pooling over the three trailing axes of a 5D 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 cubic pooling region. If an iterable, it should have three 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, 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.`**`Pool1DLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=0*, *ignore_border=True*, *mode='max'*, ***kwargs*)

1D pooling layer

Performs 1D mean or 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`.

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:

`MaxPool1DLayer` 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.`**`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.`**`Pool3DLayer`**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0, 0)*, *ignore_border=True*, *mode='max'*, ***kwargs*)

3D pooling layer

Performs 3D mean or max-pooling over the three trailing axes of a 5D 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 cubic pooling region. If an iterable, it should have three 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, 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:

`MaxPool3DLayer` 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.`**`Upscale1DLayer`**(*incoming*, *scale_factor*, *mode='repeat'*, ***kwargs*)
1D upscaling layer

Performs 1D upscaling 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.

scale_factor [integer or iterable] The scale factor. If an iterable, it should have one element.

mode [{'repeat', 'dilate'}] Upscaling mode: repeat element values or upscale leaving zeroes between upscaled elements. Default is 'repeat'.

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

class `lasagne.layers.`**`Upscale2DLayer`**(*incoming*, *scale_factor*, *mode='repeat'*, ***kwargs*)
2D upscaling layer

Performs 2D upscaling 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.

scale_factor [integer or iterable] The scale factor in each dimension. If an integer, it is promoted to a square scale factor region. If an iterable, it should have two elements.

mode [{'repeat', 'dilate'}] Upscaling mode: repeat element values or upscale leaving zeroes between upscaled elements. Default is 'repeat'.

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

Notes

Using `mode='dilate'` followed by a convolution can be realized more efficiently with a transposed convolution, see `lasagne.layers.TransposedConv2DLayer`.

class `lasagne.layers.`**`Upscale3DLayer`**(*incoming*, *scale_factor*, *mode='repeat'*, ***kwargs*)
3D upscaling layer

Performs 3D upscaling over the three trailing axes of a 5D input tensor.

Parameters

incoming [a `Layer` instance or tuple] The layer feeding into this layer, or the expected input shape.

scale_factor [integer or iterable] The scale factor in each dimension. If an integer, it is promoted to a cubic scale factor region. If an iterable, it should have three elements.

mode [{'repeat', 'dilate'}] Upscaling mode: repeat element values or upscale leaving zeroes between upscaled elements. Default is 'repeat'.

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

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.

class `lasagne.layers.`**`SpatialPyramidPoolingLayer`**(*incoming, pool_dims=[4, 2, 1], mode='max', implementation='fast', **kwargs*)

Spatial Pyramid Pooling Layer

Performs spatial pyramid pooling (SPP) over the input. It will turn a 2D input of arbitrary size into an output of fixed dimension. Hence, the convolutional part of a DNN can be connected to a dense part with a fixed number of nodes even if the dimensions of the input image are unknown.

The pooling is performed over l pooling levels. Each pooling level i will create M_i output features. M_i is given by $n_i * n_i$, with n_i as the number of pooling operation per dimension in level i, and we use a list of the n_i's as a parameter for SPP-Layer. The length of this list is the level of the spatial pyramid.

Parameters

incoming [a `Layer` instance or tuple] The layer feeding into this layer, or the expected input shape.

pool_dims [list of integers] The list of n_i's that define the output dimension of each pooling level i. The length of pool_dims is the level of the spatial pyramid.

mode [string] Pooling mode, one of 'max', 'average_inc_pad', 'average_exc_pad' Defaults to 'max'.

implementation [string] Either 'fast' or 'kaiming'. The 'fast' version uses theano's pool_2d operation, which is fast but does not work for all input sizes. The 'kaiming' mode is slower but implements the pooling as described in [1], and works with any input size.

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

Notes

This layer should be inserted between the convolutional part of a DNN and its dense part. Convolutions can be used for arbitrary input dimensions, but the size of their output will depend on their input dimensions. Connecting the output of the convolutional to the dense part then usually demands us to fix the dimensions of the network's InputLayer. The spatial pyramid pooling layer, however, allows us to leave the network input dimensions arbitrary. The advantage over a global pooling layer is the added robustness against object deformations due to the pooling on different scales.

References

[1]

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=0*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, *only_return_final=False*, ***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*, except for the first dimension: When `precompute_input == True` (the default), it must be `incoming.output_shape[0]*incoming.output_shape[1]` or `None`; when `precompute_input == False`, it must be `incoming.output_shape[0]` or `None`.

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 `Layer`] Initializer for initial hidden state (h_0).

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.

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

grad_clipping [float] If nonzero, the gradient messages are clipped to the given value during the backward pass. See [1] (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).

only_return_final [bool] If True, only return the final sequential output (e.g. for tasks where a single target value for the entire sequence is desired). In this case, Theano makes an optimization which saves memory.

References

[1]

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)`. When the hidden state of this layer is to be pre-filled (i.e. was set to a `Layer` instance) *inputs* should have length at least 2, and *inputs[-1]* is the hidden state to prefill with.

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=0*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, *only_return_final=False*, ***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 Layer] Initializer for initial hidden state (h_0).

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.

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

grad_clipping [float] If nonzero, the gradient messages are clipped to the given value during the backward pass. See [1] (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).

only_return_final [bool] If True, only return the final sequential output (e.g. for tasks where a single target value for the entire sequence is desired). In this case, Theano makes an optimization which saves memory.

References

[1]

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=0*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, *only_return_final=False*, ***kwargs*)

A long short-term memory (LSTM) layer.

Includes optional "peephole connections" and a forget gate. Based on the definition in [1], which is the current common definition. The output is computed by

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

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 Layer] Initializer for initial cell state (c_0).

hid_init [callable, np.ndarray, theano.shared or Layer] Initializer for initial hidden state (h_0).

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.

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 [float] If nonzero, the gradient messages are clipped to the given value during the backward pass. See [1] (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).

only_return_final [bool] If True, only return the final sequential output (e.g. for tasks where a single target value for the entire sequence is desired). In this case, Theano makes an optimization which saves memory.

References

[1]

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)`. When the hidden state of this layer is to be pre-filled (i.e. was set to a `Layer` instance) *inputs* should have length at least 2, and *inputs[-1]* is the hidden state to prefill with. When the cell state of this layer is to be pre-filled (i.e. was set to a `Layer` instance) *inputs* should have length at least 2, and *inputs[-1]* is the hidden state to prefill with. When both the cell state and the hidden state are being pre-filled *inputs[-2]* is the hidden state, while *inputs[-1]* is the cell state.

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=False*, *gradient_steps=-1*, *grad_clipping=0*, *unroll_scan=False*, *precompute_input=True*, *mask_input=None*, *only_return_final=False*, ***kwargs*)

Gated Recurrent Unit (GRU) Layer

Implements the recurrent step proposed in [1], 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 `Layer`] Initializer for initial hidden state (h_0).

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.

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

grad_clipping [float] If nonzero, the gradient messages are clipped to the given value during the backward pass. See [1] (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).

only_return_final [bool] If True, only return the final sequential output (e.g. for tasks where a single target value for the entire sequence is desired). In this case, Theano makes an optimization which saves memory.

Notes

An alternate update for the candidate hidden state is proposed in [2]:

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

We use the formulation from [1] because it allows us to do all matrix operations in a single dot product.

References

[1], [2], [3]

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)`. When the hidden state of this layer is to be pre-filled (i.e. was set to a `Layer` instance) *inputs* should have length at least 2, and *inputs[-1]* is the hidden state to prefill with.

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

[1]

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. [1] 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)
```

Noise layers

class `lasagne.layers.`**`DropoutLayer`**(*incoming*, *p=0.5*, *rescale=True*, *shared_axes=()*, ***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 default), scale the input by `1 / (1 - p)` when dropout is enabled, to keep the expected output mean the same.

shared_axes [tuple of int] Axes to share the dropout mask over. By default, each value can be dropped individually. `shared_axes=(0,)` uses the same mask across the batch. `shared_axes=(2, 3)` uses the same mask across the spatial dimensions of 2D feature maps.

See also:

`dropout_channels` Drops full channels of feature maps

`spatial_dropout` Alias for `dropout_channels()`

`dropout_locations` Drops full pixels or voxels of feature maps

Notes

The dropout layer is a regularizer that randomly sets input values to zero; see [1], [2] for why this might improve generalization.

The behaviour of the layer depends on the `deterministic` keyword argument passed to `lasagne.layers.get_output()`. If `True`, the layer behaves deterministically, and passes on the input unchanged. If `False` or not specified, dropout (and possibly scaling) is enabled. Usually, you would use `deterministic=False` at train time and `deterministic=True` at test time.

References

[1], [2]

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

`lasagne.layers.`**`dropout_channels`**(*incoming*, **args*, ***kwargs*)
Convenience function to drop full channels of feature maps.

Adds a `DropoutLayer` that sets feature map channels to zero, across all locations, with probability p. For convolutional neural networks, this may give better results than independent dropout [1].

Parameters

incoming [a `Layer` instance or a tuple] the layer feeding into this layer, or the expected input shape

***args, **kwargs** Any additional arguments and keyword arguments are passed on to the `DropoutLayer` constructor, except for *shared_axes*.

Returns

layer [`DropoutLayer` instance] The dropout layer with *shared_axes* set to drop channels.

References

[1]

lasagne.layers.**spatial_dropout**(*incoming*, **args*, ***kwargs*)
alias of dropout_channels()

lasagne.layers.**dropout_locations**(*incoming*, **args*, ***kwargs*)
Convenience function to drop full locations of feature maps.

Adds a DropoutLayer that sets feature map locations (i.e., pixels or voxels) to zero, across all channels, with probability p.

Parameters

incoming [a Layer instance or a tuple] the layer feeding into this layer, or the expected input shape

***args, **kwargs** Any additional arguments and keyword arguments are passed on to the DropoutLayer constructor, except for *shared_axes*.

Returns

layer [DropoutLayer instance] The dropout layer with *shared_axes* set to drop locations.

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

Add zero-mean Gaussian noise of given standard deviation to the input [1].

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

[1]

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

Parameters

input [tensor] output from the previous layer

deterministic [bool] If true noise is disabled, see notes

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)
```

Merge layers

class lasagne.layers.**ConcatLayer**(*incomings*, *axis=1*, *cropping=None*, ***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

cropping [None or [crop]] Cropping for each input axis. Cropping is described in the docstring for autocrop(). Cropping is always disabled for *axis*.

lasagne.layers.**concat**

alias of ConcatLayer

class lasagne.layers.**ElemwiseMergeLayer**(*incomings*, *merge_function*, *cropping=None*, ***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.

cropping [None or [crop]] Cropping for each input axis. Cropping is described in the docstring for autocrop()

See also:

ElemwiseSumLayer Shortcut for sum layer.

class lasagne.layers.**ElemwiseSumLayer**(*incomings*, *coeffs=1*, *cropping=None*, ***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.

cropping [None or [crop]] Cropping for each input axis. Cropping is described in the docstring for autocrop()

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.)

Normalization layers

The LocalResponseNormalization2DLayer implementation contains code from pylearn2, which is covered by the following license:

class lasagne.layers.**LocalResponseNormalization2DLayer**(*incoming*, *alpha=0.0001*, *k=2*, *beta=0.75*, *n=5*, ***kwargs*)

Cross-channel Local Response Normalization for 2D feature maps.

Aggregation is purely across channels, not within channels, and performed "pixelwise".

If the value of the i th channel is x_i, the output is

$$x_i = \frac{x_i}{(k + (\alpha \sum_j x_j^2))^\beta}$$

where the summation is performed over this position on n neighboring channels.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape. Must follow *BC01* layout, i.e., `(batchsize, channels, rows, columns)`.

alpha [float scalar] coefficient, see equation above

k [float scalar] offset, see equation above

beta [float scalar] exponent, see equation above

n [int] number of adjacent channels to normalize over, must be odd

Notes

This code is adapted from pylearn2. See the module docstring for license information.

class `lasagne.layers.`**`BatchNormLayer`**(*incoming*, *axes='auto'*, *epsilon=1e-4*, *alpha=0.1*, *beta=lasagne.init.Constant(0)*, *gamma=lasagne.init.Constant(1)*, *mean=lasagne.init.Constant(0)*, *inv_std=lasagne.init.Constant(1)*, ***kwargs*)

Batch Normalization

This layer implements batch normalization of its inputs, following [1]:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta$$

That is, the input is normalized to zero mean and unit variance, and then linearly transformed. The crucial part is that the mean and variance are computed across the batch dimension, i.e., over examples, not per example.

During training, μ and σ^2 are defined to be the mean and variance of the current input mini-batch x, and during testing, they are replaced with average statistics over the training data. Consequently, this layer has four stored parameters: β, γ, and the averages μ and σ^2 (nota bene: instead of σ^2, the layer actually stores $1/\sqrt{\sigma^2 + \epsilon}$, for compatibility to cuDNN). By default, this layer learns the average statistics as exponential moving averages computed during training, so it can be plugged into an existing network without any changes of the training procedure (see Notes).

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

axes ['auto', int or tuple of int] The axis or axes to normalize over. If `'auto'` (the default), normalize over all axes except for the second: this will normalize over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers.

epsilon [scalar] Small constant ϵ added to the variance before taking the square root and dividing by it, to avoid numerical problems

alpha [scalar] Coefficient for the exponential moving average of batch-wise means and standard deviations computed during training; the closer to one, the more it will depend on the last batches seen

beta [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for β. Must match the incoming shape, skipping all axes in *axes*. Set to `None` to fix it to 0.0 instead of learning it. See `lasagne.utils.create_param()` for more information.

gamma [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for γ. Must match the incoming shape, skipping all axes in *axes*. Set to `None` to fix it to 1.0 instead of learning it. See `lasagne.utils.create_param()` for more information.

mean [Theano shared variable, expression, numpy array, or callable] Initial value, expression or initializer for μ. Must match the incoming shape, skipping all axes in *axes*. See `lasagne.utils.create_param()` for more information.

inv_std [Theano shared variable, expression, numpy array, or callable] Initial value, expression or initializer for $1/\sqrt{\sigma^2 + \epsilon}$. Must match the incoming shape, skipping all axes in *axes*. See `lasagne.utils.create_param()` for more information.

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

See also:

`batch_norm` Convenience function to apply batch normalization to a layer

Notes

This layer should be inserted between a linear transformation (such as a `DenseLayer`, or `Conv2DLayer`) and its nonlinearity. The convenience function `batch_norm()` modifies an existing layer to insert batch normalization in front of its nonlinearity.

The behavior can be controlled by passing keyword arguments to `lasagne.layers.get_output()` when building the output expression of any network containing this layer.

During training, [1] normalize each input mini-batch by its statistics and update an exponential moving average of the statistics to be used for validation. This can be achieved by passing `deterministic=False`. For validation, [1] normalize each input mini-batch by the stored statistics. This can be achieved by passing `deterministic=True`.

For more fine-grained control, `batch_norm_update_averages` can be passed to update the exponential moving averages (`True`) or not (`False`), and `batch_norm_use_averages` can be passed to use the exponential moving averages for normalization (`True`) or normalize each mini-batch by its own statistics (`False`). These settings override `deterministic`.

Note that for testing a model after training, [1] replace the stored exponential moving average statistics by fixing all network weights and re-computing average statistics over the training data in a layerwise fashion. This is not part of the layer implementation.

In case you set *axes* to not include the batch dimension (the first axis, usually), normalization is done per example, not across examples. This does not require any averages, so you can pass `batch_norm_update_averages` and `batch_norm_use_averages` as `False` in this case.

References

[1]

`lasagne.layers.`**`batch_norm`**(*layer*, ***kwargs*)

Apply batch normalization to an existing layer. This is a convenience function modifying an existing layer to include batch normalization: It will steal the layer's nonlinearity if there is one (effectively introducing

the normalization right before the nonlinearity), remove the layer's bias if there is one (because it would be redundant), and add a `BatchNormLayer` and `NonlinearityLayer` on top.

Parameters

layer [A `Layer` instance] The layer to apply the normalization to; note that it will be irreversibly modified as specified above

****kwargs** Any additional keyword arguments are passed on to the `BatchNormLayer` constructor.

Returns

BatchNormLayer or NonlinearityLayer instance A batch normalization layer stacked on the given modified *layer*, or a nonlinearity layer stacked on top of both if *layer* was nonlinear.

Examples

Just wrap any layer into a `batch_norm()` call on creating it:

```
>>> from lasagne.layers import InputLayer, DenseLayer, batch_norm
>>> from lasagne.nonlinearities import tanh
>>> l1 = InputLayer((64, 768))
>>> l2 = batch_norm(DenseLayer(l1, num_units=500, nonlinearity=tanh))
```

This introduces batch normalization right before its nonlinearity:

```
>>> from lasagne.layers import get_all_layers
>>> [l.__class__.__name__ for l in get_all_layers(l2)]
['InputLayer', 'DenseLayer', 'BatchNormLayer', 'NonlinearityLayer']
```

class `lasagne.layers.`**`StandardizationLayer`**(*incoming*, *axes='auto'*, *epsilon=0.0001*, ***kwargs*)

Standardize inputs to zero mean and unit variance:

$$y_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

The mean μ_i and variance σ_i^2 are computed and shared across a given set of axes. In contrast to batch normalization, these axes usually do not include the batch dimension, so each example is normalized independently from other examples in the minibatch, both during training and testing.

The `StandardizationLayer` can be employed to realize instance normalization [1] and layer normalization [2], for both of which convenience functions (`instance_norm()` and `layer_norm()`) are available.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

axes ['auto', 'spatial', 'features', int or tuple of int] The axis or axes to normalize over. If `'auto'` (the default), two-dimensional inputs are normalized over the last dimension (i.e., this will normalize over units for dense layers), input tensors with more than two dimensions are normalized over all but the first two dimensions (i.e., this will normalize over all spatial dimensions for convolutional layers). If `'spatial'`, will normalize over all but the first two dimensions. If `'features'`, will normalize over all but the first dimension.

epsilon [scalar] Small constant ϵ added to the variance before taking the square root and dividing by it, to avoid numerical problems

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

See also:

`instance_norm` Convenience function to apply instance normalization

`layer_norm` Convenience function to apply layer normalization to a layer

References

[1], [2]

lasagne.layers.**instance_norm**(*layer*, *learn_scale=True*, *learn_bias=True*, ***kwargs*)

Apply instance normalization to an existing layer. This is a convenience function modifying an existing layer to include instance normalization: It will steal the layer's nonlinearity if there is one (effectively introducing the normalization right before the nonlinearity), remove the layer's bias if there is one (because it would be effectless), and add a `StandardizationLayer` and `NonlinearityLayer` on top. Depending on the given arguments, an additional `ScaleLayer` and `BiasLayer` will be inserted in between.

In effect, it will separately standardize each feature map of each input example, followed by an optional scale and shift learned per channel, followed by the original nonlinearity, as proposed in [1].

Parameters

layer [A `Layer` instance] The layer to apply the normalization to; note that it will be irreversibly modified as specified above

learn_scale [bool (default: True)] Whether to add a ScaleLayer after the StandardizationLayer

learn_bias [bool (default: True)] Whether to add a BiasLayer after the StandardizationLayer (or the optional ScaleLayer)

****kwargs** Any additional keyword arguments are passed on to the `StandardizationLayer` constructor.

Returns

StandardizationLayer, ScaleLayer, BiasLayer, or NonlinearityLayer instance The last layer stacked on top of the given modified *layer* to implement instance normalization with optional scaling and shifting.

References

[1]

Examples

Just wrap any layer into a `instance_norm()` call on creating it:

```
>>> from lasagne.layers import InputLayer, Conv2DLayer, instance_norm
>>> from lasagne.nonlinearities import rectify
>>> l1 = InputLayer((10, 3, 28, 28))
>>> l2 = instance_norm(Conv2DLayer(l1, num_filters=64, filter_size=3,
...                                nonlinearity=rectify))
```

This introduces instance normalization right before its nonlinearity:

```
>>> from lasagne.layers import get_all_layers
>>> [l.__class__.__name__ for l in get_all_layers(l2)]
['InputLayer', 'Conv2DLayer', 'StandardizationLayer', 'ScaleLayer', 'BiasLayer', 'NonlinearityLa
```

lasagne.layers.**layer_norm**(*layer*, ***kwargs*)

Apply layer normalization to an existing layer. This is a convenience function modifying an existing layer to include layer normalization: It will steal the layer's nonlinearity if there is one (effectively introducing the normalization right before the nonlinearity), remove the layer's bias if there is one, and add a StandardizationLayer, ScaleLayer, BiasLayer, and NonlinearityLayer on top.

In effect, it will standardize each input example across the feature and spatial dimensions (if any), followed by a scale and shift learned per feature, followed by the original nonlinearity, as proposed in [1].

Parameters

layer [A Layer instance] The layer to apply the normalization to; note that it will be irreversibly modified as specified above

****kwargs** Any additional keyword arguments are passed on to the StandardizationLayer constructor.

Returns

StandardizationLayer or NonlinearityLayer instance The last layer stacked on top of the given modified *layer* to implement layer normalization with feature-wise scaling and shifting.

References

[1]

Examples

Just wrap any layer into a layer_norm() call on creating it:

```
>>> from lasagne.layers import InputLayer, DenseLayer, layer_norm
>>> from lasagne.nonlinearities import rectify
>>> l1 = InputLayer((10, 28))
>>> l2 = layer_norm(DenseLayer(l1, num_units=64, nonlinearity=rectify))
```

This introduces layer normalization right before its nonlinearity:

```
>>> from lasagne.layers import get_all_layers
>>> [l.__class__.__name__ for l in get_all_layers(l2)]
['InputLayer', 'DenseLayer', 'StandardizationLayer', 'ScaleLayer', 'BiasLayer', 'NonlinearityLay
```

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, expression, numpy array or callable] Initial value, expression or initializer for the embedding matrix. This should be a matrix with shape `(input_size, output_size)`. See `lasagne.utils.create_param()` for more information.

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)
```

Special-purpose layers

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.`**`BiasLayer`**(*incoming*, *b=lasagne.init.Constant(0)*, *shared_axes='auto'*, ***kwargs*)

A layer that just adds a (trainable) bias term.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases and pass through its input unchanged. Otherwise, the bias shape must match the incoming shape, skipping those axes the biases are shared over (see the example below). See `lasagne.utils.create_param()` for more information.

shared_axes ['auto', int or tuple of int] The axis or axes to share biases over. If `'auto'` (the default), share over all axes except for the second: this will share biases over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers.

Notes

The bias parameter dimensionality is the input dimensionality minus the number of axes the biases are shared over, which matches the bias parameter conventions of `DenseLayer` or `Conv2DLayer`. For example:

```
>>> layer = BiasLayer((20, 30, 40, 50), shared_axes=(0, 2))
>>> layer.b.get_value().shape
(30, 50)
```

class `lasagne.layers.`**`ScaleLayer`**(*incoming*, *scales=lasagne.init.Constant(1)*, *shared_axes='auto'*, ***kwargs*)

A layer that scales its inputs by learned coefficients.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

scales [Theano shared variable, expression, numpy array, or callable] Initial value, expression or initializer for the scale. The scale shape must match the incoming shape, skipping those axes the scales are shared over (see the example below). See `lasagne.utils.create_param()` for more information.

shared_axes ['auto', int or tuple of int] The axis or axes to share scales over. If `'auto'` (the default), share over all axes except for the second: this will share scales over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers.

Notes

The scales parameter dimensionality is the input dimensionality minus the number of axes the scales are shared over, which matches the bias parameter conventions of `DenseLayer` or `Conv2DLayer`. For example:

```
>>> layer = ScaleLayer((20, 30, 40, 50), shared_axes=(0, 2))
>>> layer.scales.get_value().shape
(30, 50)
```

`lasagne.layers.`**`standardize`**(*layer*, *offset*, *scale*, *shared_axes='auto'*)

Convenience function for standardizing inputs by applying a fixed offset and scale. This is usually useful when you want the input to your network to, say, have zero mean and unit standard deviation over the feature dimensions. This layer allows you to include the appropriate statistics to achieve this normalization as part of your network, and applies them to its input. The statistics are supplied as the *offset* and *scale* parameters, which are applied to the input by subtracting *offset* and dividing by *scale*, sharing dimensions as specified by the *shared_axes* argument.

Parameters

layer [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape.

offset [Theano shared variable or numpy array] The offset to apply (via subtraction) to the axis/axes being standardized.

scale [Theano shared variable or numpy array] The scale to apply (via division) to the axis/axes being standardized.

shared_axes ['auto', int or tuple of int] The axis or axes to share the offset and scale over. If `'auto'` (the default), share over all axes except for the second: this will share scales over

the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers.

Examples

Assuming your training data exists in a 2D numpy ndarray called `training_data`, you can use this function to scale input features to the [0, 1] range based on the training set statistics like so:

```
>>> import lasagne
>>> import numpy as np
>>> training_data = np.random.standard_normal((100, 20))
>>> input_shape = (None, training_data.shape[1])
>>> l_in = lasagne.layers.InputLayer(input_shape)
>>> offset = training_data.min(axis=0)
>>> scale = training_data.max(axis=0) - training_data.min(axis=0)
>>> l_std = standardize(l_in, offset, scale, shared_axes=0)
```

Alternatively, to z-score your inputs based on training set statistics, you could set `offset = training_data.mean(axis=0)` and `scale = training_data.std(axis=0)` instead.

class `lasagne.layers.`**`ExpressionLayer`**(*incoming*, *function*, *output_shape=None*, ***kwargs*)

This layer provides boilerplate for a custom layer that applies a simple transformation to the input.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape.

function [callable] A function to be applied to the output of the previous layer.

output_shape [None, callable, tuple, or 'auto'] Specifies the output shape of this layer. If a tuple, this fixes the output shape for any input shape (the tuple can contain None if some dimensions may vary). If a callable, it should return the calculated output shape given the input shape. If None, the output shape is assumed to be the same as the input shape. If 'auto', an attempt will be made to automatically infer the correct output shape.

Notes

An `ExpressionLayer` that does not change the shape of the data (i.e., is constructed with the default setting of `output_shape=None`) is functionally equivalent to a `NonlinearityLayer`.

Examples

```
>>> from lasagne.layers import InputLayer, ExpressionLayer
>>> l_in = InputLayer((32, 100, 20))
>>> l1 = ExpressionLayer(l_in, lambda X: X.mean(-1), output_shape='auto')
>>> l1.output_shape
(32, 100)
```

class `lasagne.layers.`**`InverseLayer`**(*incoming*, *layer*, ***kwargs*)

The `InverseLayer` class performs inverse operations for a single layer of a neural network by applying the partial derivative of the layer to be inverted with respect to its input: transposed layer for a `DenseLayer`, deconvolutional layer for `Conv2DLayer`, `Conv1DLayer`; or an unpooling layer for `MaxPool2DLayer`.

It is specially useful for building (convolutional) autoencoders with tied parameters.

Note that if the layer to be inverted contains a nonlinearity and/or a bias, the `InverseLayer` will include the derivative of that in its computation.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape.

layer [a `Layer` instance or a tuple] The layer with respect to which the instance of the `InverseLayer` is inverse to.

Examples

```
>>> import lasagne
>>> from lasagne.layers import InputLayer, Conv2DLayer, DenseLayer
>>> from lasagne.layers import InverseLayer
>>> l_in = InputLayer((100, 3, 28, 28))
>>> l1 = Conv2DLayer(l_in, num_filters=16, filter_size=5)
>>> l2 = DenseLayer(l1, num_units=20)
>>> l_u2 = InverseLayer(l2, l2)  # backprop through l2
>>> l_u1 = InverseLayer(l_u2, l1)  # backprop through l1
```

class `lasagne.layers.`**`TransformerLayer`**(*incoming*, *localization_network*, *downsample_factor=1*, *border_mode='nearest'*, ***kwargs*)

Spatial transformer layer

The layer applies an affine transformation on the input. The affine transformation is parameterized with six learned parameters [1]. The output is interpolated with a bilinear transformation.

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)`.

localization_network [a `Layer` instance] The network that calculates the parameters of the affine transformation. See the example for how to initialize to the identity transform.

downsample_factor [float or iterable of float] A float or a 2-element tuple specifying the downsample factor for the output image (in both spatial dimensions). A value of 1 will keep the original size of the input. Values larger than 1 will downsample the input. Values below 1 will upsample the input.

border_mode ['nearest', 'mirror', or 'wrap'] Determines how border conditions are handled during interpolation. If 'nearest', points outside the grid are clipped to the boundary. If 'mirror', points are mirrored across the boundary. If 'wrap', points wrap around to the other side of the grid. See http://stackoverflow.com/q/22669252/22670830#22670830 for details.

References

[1]

Examples

Here we set up the layer to initially do the identity transform, similarly to [1]. Note that you will want to use a localization with linear output. If the output from the localization networks is [t1, t2, t3, t4, t5, t6] then t1 and t5 determines zoom, t2 and t4 determines skewness, and t3 and t6 move the center position.

```
>>> import numpy as np
>>> import lasagne
>>> b = np.zeros((2, 3), dtype='float32')
>>> b[0, 0] = 1
>>> b[1, 1] = 1
>>> b = b.flatten()  # identity transform
>>> W = lasagne.init.Constant(0.0)
>>> l_in = lasagne.layers.InputLayer((None, 3, 28, 28))
>>> l_loc = lasagne.layers.DenseLayer(l_in, num_units=6, W=W, b=b,
... nonlinearity=None)
>>> l_trans = lasagne.layers.TransformerLayer(l_in, l_loc)
```

class lasagne.layers.**TPSTransformerLayer**(*incoming*, *localization_network*, *downsample_factor=1*, *control_points=16*, *precompute_grid='auto'*, *border_mode='nearest'*, ***kwargs*)

Spatial transformer layer

The layer applies a thin plate spline transformation [2] on the input as in [1]. The thin plate spline transform is determined based on the movement of some number of control points. The starting positions for these control points are fixed. The output is interpolated with a bilinear transformation.

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)`.

localization_network [a Layer instance] The network that calculates the parameters of the thin plate spline transformation as the x and y coordinates of the destination offsets of each control point. The output of the localization network should be a 2D tensor, with shape `(batch_size, 2 * num_control_points)`

downsample_factor [float or iterable of float] A float or a 2-element tuple specifying the downsample factor for the output image (in both spatial dimensions). A value of 1 will keep the original size of the input. Values larger than 1 will downsample the input. Values below 1 will upsample the input.

control_points [integer] The number of control points to be used for the thin plate spline transformation. These points will be arranged as a grid along the image, so the value must be a perfect square. Default is 16.

precompute_grid ['auto' or boolean] Flag to precompute the U function [2] for the grid and source points. If 'auto', will be set to true as long as the input height and width are specified. If true, the U function is computed when the layer is constructed for a fixed input shape. If false, grid will be computed as part of the Theano computational graph, which is substantially slower as this computation scales with num_pixels*num_control_points. Default is 'auto'.

border_mode ['nearest', 'mirror', or 'wrap'] Determines how border conditions are handled during interpolation. If 'nearest', points outside the grid are clipped to the boundary'. If 'mirror', points are mirrored across the boundary. If 'wrap', points wrap around to the other side of the grid. See http://stackoverflow.com/q/22669252/22670830#22670830 for details.

References

[1], [2]

Examples

Here, we'll implement an identity transform using a thin plate spline transform. First we'll create the destination control point offsets. To make everything invariant to the shape of the image, the x and y range of the image is normalized to [-1, 1] as in ref [1]. To replicate an identity transform, we'll set the bias to have all offsets be 0. More complicated transformations can easily be implemented using different x and y offsets (importantly, each control point can have it's own pair of offsets).

```
>>> import numpy as np
>>> import lasagne
>>>
>>> # Create the network
>>> # we'll initialize the weights and biases to zero, so it starts
>>> # as the identity transform (all control point offsets are zero)
>>> W = b = lasagne.init.Constant(0.0)
>>>
>>> # Set the number of points
>>> num_points = 16
>>>
>>> l_in = lasagne.layers.InputLayer((None, 3, 28, 28))
>>> l_loc = lasagne.layers.DenseLayer(l_in, num_units=2*num_points,
...                                   W=W, b=b, nonlinearity=None)
>>> l_trans = lasagne.layers.TPSTransformerLayer(l_in, l_loc,
...                                              control_points=num_points)
```

class lasagne.layers.**ParametricRectifierLayer**(*incoming*, *alpha=init.Constant(0.25)*, *shared_axes='auto'*, ***kwargs*)

A layer that applies parametric rectify nonlinearity to its input following [1].

Equation for the parametric rectifier linear unit: $\varphi(x) = \max(x, 0) + \alpha \min(x, 0)$

Parameters

incoming [a Layer instance or a tuple] The layer feeding into this layer, or the expected input shape

alpha [Theano shared variable, expression, numpy array or callable] Initial value, expression or initializer for the alpha values. The shape must match the incoming shape, skipping those axes the alpha values are shared over (see the example below). See lasagne.utils.create_param() for more information.

shared_axes ['auto', 'all', int or tuple of int] The axes along which the parameters of the rectifier units are going to be shared. If 'auto' (the default), share over all axes except for the second - this will share the parameter over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers. If 'all', share over all axes, which corresponds to a single scalar parameter.

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

Notes

The alpha parameter dimensionality is the input dimensionality minus the number of axes it is shared over, which matches the same convention as the BiasLayer.

```
>>> layer = ParametricRectifierLayer((20, 3, 28, 28), shared_axes=(0, 3))
>>> layer.alpha.get_value().shape
(3, 28)
```

References

[1]

lasagne.layers.**prelu**(*layer*, ***kwargs*)

Convenience function to apply parametric rectify to a given layer's output. Will set the layer's nonlinearity to identity if there is one and will apply the parametric rectifier instead.

Parameters

layer: a :class:'Layer' instance The *Layer* instance to apply the parametric rectifier layer to; note that it will be irreversibly modified as specified above

****kwargs** Any additional keyword arguments are passed to the `ParametericRectifierLayer`

Examples

Note that this function modifies an existing layer, like this:

```
>>> from lasagne.layers import InputLayer, DenseLayer, prelu
>>> layer = InputLayer((32, 100))
>>> layer = DenseLayer(layer, num_units=200)
>>> layer = prelu(layer)
```

In particular, `prelu()` can *not* be passed as a nonlinearity.

class lasagne.layers.**RandomizedRectifierLayer**(*incoming*, *lower=0.3*, *upper=0.8*, *shared_axes='auto'*, ***kwargs*)

A layer that applies a randomized leaky rectify nonlinearity to its input.

The randomized leaky rectifier was first proposed and used in the Kaggle NDSB Competition, and later evaluated in [1]. Compared to the standard leaky rectifier `leaky_rectify()`, it has a randomly sampled slope for negative input during training, and a fixed slope during evaluation.

Equation for the randomized rectifier linear unit during training: $\varphi(x) = \max((\sim U(lower, upper)) \cdot x, x)$

During evaluation, the factor is fixed to the arithmetic mean of *lower* and *upper*.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

lower [Theano shared variable, expression, or constant] The lower bound for the randomly chosen slopes.

upper [Theano shared variable, expression, or constant] The upper bound for the randomly chosen slopes.

shared_axes ['auto', 'all', int or tuple of int] The axes along which the random slopes of the rectifier units are going to be shared. If `'auto'` (the default), share over all axes except for the second - this will share the random slope over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers. If `'all'`, share over all axes, thus using a single random slope.

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

References

[1]

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

Parameters

input [tensor] output from the previous layer

deterministic [bool] If true, the arithmetic mean of lower and upper are used for the leaky slope.

lasagne.layers.**rrelu**(*layer*, ***kwargs*)

Convenience function to apply randomized rectify to a given layer's output. Will set the layer's nonlinearity to identity if there is one and will apply the randomized rectifier instead.

Parameters

layer: a :class:'Layer' instance The *Layer* instance to apply the randomized rectifier layer to; note that it will be irreversibly modified as specified above

****kwargs** Any additional keyword arguments are passed to the RandomizedRectifierLayer

Examples

Note that this function modifies an existing layer, like this:

```
>>> from lasagne.layers import InputLayer, DenseLayer, rrelu
>>> layer = InputLayer((32, 100))
>>> layer = DenseLayer(layer, num_units=200)
>>> layer = rrelu(layer)
```

In particular, `rrelu()` can *not* be passed as a nonlinearity.

lasagne.layers.corrmm

This module houses layers that require a GPU to work. Its layers are not automatically imported into the `lasagne.layers` namespace: To use these layers, you need to `import lasagne.layers.corrmm` explicitly.

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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 4D tensor with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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

Attributes

W [Theano shared variable] Variable representing the filter weights.

b [Theano shared variable] Variable representing the biases.

lasagne.layers.cuda_convnet

This module houses layers that require *pylearn2 <https://deeplearning.net/software/pylearn2>* to work. Its layers are not automatically imported into the `lasagne.layers` namespace: To use these layers, you need to `import lasagne.layers.cuda_convnet` explicitly.

lasagne.layers.dnn

This module houses layers that require cuDNN to work. Its layers are not automatically imported into the `lasagne.layers` namespace: To use these layers, you need to `import lasagne.layers.dnn` explicitly.

Note that these layers are not required to use cuDNN: If cuDNN is available, Theano will use it for the default convolution and pooling layers anyway. However, they allow you to enforce the usage of cuDNN or use features not available in `lasagne.layers`.

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.**Pool3DDNNLayer**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0, 0)*, *ignore_border=True*, *mode='max'*, ***kwargs*)

3D pooling layer

Performs 3D mean- or max-pooling over the 3 trailing axes of a 5D 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.

class lasagne.layers.dnn.**MaxPool3DDNNLayer**(*incoming*, *pool_size*, *stride=None*, *pad=(0, 0, 0)*, *ignore_border=True*, ***kwargs*)

3D max-pooling layer

Subclass of Pool3DDNNLayer fixing mode='max', provided for consistency to 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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 4D tensor with shape `(num_filters, num_input_channels, filter_rows, filter_columns)`. See `lasagne.utils.create_param()` for more information.

b [Theano shared variable, expression, numpy array, callable or `None`] Initial value, expression or initializer for the biases. If set to `None`, the layer will have no biases. Otherwise, biases should be 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, output_rows, output_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.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of

channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

class `lasagne.layers.dnn.`**`Conv3DDNNLayer`**(*incoming*, *num_filters*, *filter_size*, *stride=(1, 1, 1)*, *pad=0*, *untie_biases=False*, *W=lasagne.init.GlorotUniform()*, *b=lasagne.init.Constant(0.)*, *nonlinearity=lasagne.nonlinearities.rectify*, *flip_filters=False*, ***kwargs*)

3D convolutional layer

Performs a 3D convolution on its input and optionally adds a bias and applies an elementwise nonlinearity. This implementation uses `theano.sandbox.cuda.dnn.dnn_conv3d` 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 5D tensor, with shape `(batch_size, num_input_channels, input_depth, 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 3-element tuple specifying the size of the filters.

stride [int or iterable of int] An integer or a 3-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 three 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 (rounded down) on both sides. When `stride=1` this results in an output size equal to the input size. Even filter size is not supported.

`'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 4D tensor.

W [Theano shared variable, expression, numpy array or callable] Initial value, expression or initializer for the weights. These should be a 5D tensor with shape

(num_filters, num_input_channels, filter_depth, filter_rows, filter_columns). See lasagne.utils.create_param() for more information.

b [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for the biases. If set to None, the layer will have no biases. Otherwise, biases should be 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, output_depth, output_rows, output_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 learned, but if you want to compute predictions with pre-trained weights, take care if they need flipping.

num_groups [int (default: 1)] The number of groups to split the input channels and output channels into, such that data does not cross the group boundaries. Requires the number of channels to be divisible by the number of groups, and requires Theano 0.10 or later for more than one group.

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

Attributes

W [Theano shared variable or expression] Variable or expression representing the filter weights.

b [Theano shared variable or expression] Variable or expression representing the biases.

class lasagne.layers.dnn.**SpatialPyramidPoolingDNNLayer**(*incoming, pool_dims=[4, 2, 1], mode='max', **kwargs*)

Spatial Pyramid Pooling Layer

Performs spatial pyramid pooling (SPP) over the input. It will turn a 2D input of arbitrary size into an output of fixed dimension. Hence, the convolutional part of a DNN can be connected to a dense part with a fixed number of nodes even if the dimensions of the input image are unknown.

The pooling is performed over l pooling levels. Each pooling level i will create M_i output features. M_i is given by $n_i * n_i$, with n_i as the number of pooling operation per dimension in level i, and we use a list of the n_i's as a parameter for SPP-Layer. The length of this list is the level of the spatial pyramid.

Parameters

incoming [a Layer instance or tuple] The layer feeding into this layer, or the expected input shape.

pool_dims [list of integers] The list of n_i's that define the output dimension of each pooling level i. The length of pool_dims is the level of the spatial pyramid.

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

This layer should be inserted between the convolutional part of a DNN and its dense part. Convolutions can be used for arbitrary input dimensions, but the size of their output will depend on their input dimensions. Connecting the output of the convolutional to the dense part then usually demands us to fix the dimensions of the network's InputLayer. The spatial pyramid pooling layer, however, allows us to leave the network input dimensions arbitrary. The advantage over a global pooling layer is the added robustness against object deformations due to the pooling on different scales.

References

[1]

class `lasagne.layers.dnn.`**`BatchNormDNNLayer`**(*incoming*, *axes='auto'*, *epsilon=1e-4*, *alpha=0.1*, *beta=lasagne.init.Constant(0)*, *gamma=lasagne.init.Constant(1)*, *mean=lasagne.init.Constant(0)*, *inv_std=lasagne.init.Constant(1)*, ***kwargs*)

Batch Normalization

This layer implements batch normalization of its inputs:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta$$

This is a drop-in replacement for `lasagne.layers.BatchNormLayer` that uses cuDNN for improved performance and reduced memory usage.

Parameters

incoming [a `Layer` instance or a tuple] The layer feeding into this layer, or the expected input shape

axes ['auto', int or tuple of int] The axis or axes to normalize over. If `'auto'` (the default), normalize over all axes except for the second: this will normalize over the minibatch dimension for dense layers, and additionally over all spatial dimensions for convolutional layers. Only supports `'auto'` and the equivalent axes list, or `0` and `(0,)` to normalize over the minibatch dimension only.

epsilon [scalar] Small constant ϵ added to the variance before taking the square root and dividing by it, to avoid numerical problems. Must not be smaller than `1e-5`.

alpha [scalar] Coefficient for the exponential moving average of batch-wise means and standard deviations computed during training; the closer to one, the more it will depend on the last batches seen

beta [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for β. Must match the incoming shape, skipping all axes in *axes*. Set to `None` to fix it to 0.0 instead of learning it. See `lasagne.utils.create_param()` for more information.

gamma [Theano shared variable, expression, numpy array, callable or None] Initial value, expression or initializer for γ. Must match the incoming shape, skipping all axes in *axes*. Set to `None` to fix it to 1.0 instead of learning it. See `lasagne.utils.create_param()` for more information.

mean [Theano shared variable, expression, numpy array, or callable] Initial value, expression or initializer for μ. Must match the incoming shape, skipping all axes in *axes*. See `lasagne.utils.create_param()` for more information.

inv_std [Theano shared variable, expression, numpy array, or callable] Initial value, expression or initializer for $1/\sqrt{\sigma^2 + \epsilon}$. Must match the incoming shape, skipping all axes in *axes*. See `lasagne.utils.create_param()` for more information.

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

See also:

batch_norm_dnn Convenience function to apply batch normalization

Notes

This layer should be inserted between a linear transformation (such as a `DenseLayer`, or `Conv2DLayer`) and its nonlinearity. The convenience function `batch_norm_dnn()` modifies an existing layer to insert cuDNN batch normalization in front of its nonlinearity.

For further information, see `lasagne.layers.BatchNormLayer`. This implementation is fully compatible, except for restrictions on the *axes* and *epsilon* arguments.

`lasagne.layers.dnn.`**batch_norm_dnn**(*layer*, ***kwargs*)

Apply cuDNN batch normalization to an existing layer. This is a drop-in replacement for `lasagne.layers.batch_norm()`; see there for further information.

Parameters

layer [A `Layer` instance] The layer to apply the normalization to; note that it will be modified as specified in `lasagne.layers.batch_norm()`

****kwargs** Any additional keyword arguments are passed on to the `BatchNormDNNLayer` constructor.

Returns

BatchNormDNNLayer or NonlinearityLayer instance A batch normalization layer stacked on the given modified *layer*, or a nonlinearity layer stacked on top of both if *layer* was nonlinear.

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`	Returns a list of Theano shared variables or expressions that parameterize the layer.
`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

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.
NINLayer	Network-in-network layer.

Convolutional layers

Conv1DLayer	1D convolutional layer
Conv2DLayer	2D convolutional layer
Conv3DLayer	3D convolutional layer
TransposedConv2DLayer	2D transposed convolution layer
Deconv2DLayer	alias of TransposedConv2DLayer
DilatedConv2DLayer	2D dilated convolution layer

Local layers

LocallyConnected2DLayer	2D locally connected layer

Pooling layers

MaxPool1DLayer	1D max-pooling layer
MaxPool2DLayer	2D max-pooling layer
MaxPool3DLayer	3D max-pooling layer
Pool1DLayer	1D pooling layer
Pool2DLayer	2D pooling layer
Pool3DLayer	3D pooling layer
Upscale1DLayer	1D upscaling layer
Upscale2DLayer	2D upscaling layer
Upscale3DLayer	3D upscaling layer
GlobalPoolLayer	Global pooling layer
FeaturePoolLayer	Feature pooling layer
FeatureWTALayer	'Winner Take All' layer
SpatialPyramidPoolingLayer	Spatial Pyramid Pooling Layer

Recurrent layers

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

Noise layers

DropoutLayer	Dropout layer
dropout	alias of DropoutLayer
dropout_channels	Convenience function to drop full channels of feature maps.
spatial_dropout	Convenience function to drop full channels of feature maps.
dropout_locations	Convenience function to drop full locations of feature maps.
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.

Normalization layers

LocalResponseNormalization2DLayer	Cross-channel Local Response Normalization for 2D feature maps.
BatchNormLayer	Batch Normalization
batch_norm	Apply batch normalization to an existing layer.
StandardizationLayer	Standardize inputs to zero mean and unit variance:
instance_norm	Apply instance normalization to an existing layer.
layer_norm	Apply layer normalization to an existing layer.

Embedding layers

EmbeddingLayer	A layer for word embeddings.

Special-purpose layers

NonlinearityLayer	A layer that just applies a nonlinearity.

BiasLayer	A layer that just adds a (trainable) bias term.
ScaleLayer	A layer that scales its inputs by learned coefficients.
standardize	Convenience function for standardizing inputs by applying a fixed offset and scale.
ExpressionLayer	This layer provides boilerplate for a custom layer that applies a simple transformation to the inp
InverseLayer	The InverseLayer class performs inverse operations for a single layer of a neural network l
TransformerLayer	Spatial transformer layer
TPSTransformerLayer	Spatial transformer layer
ParametricRectifierLayer	A layer that applies parametric rectify nonlinearity to its input following [R6ee2c8fdcfdf-1].
prelu	Convenience function to apply parametric rectify to a given layer's output.
RandomizedRectifierLayer	A layer that applies a randomized leaky rectify nonlinearity to its input.
rrelu	Convenience function to apply randomized rectify to a given layer's output.

lasagne.layers.corrmm

corrmm.Conv2DMMLayer	2D convolutional layer

lasagne.layers.cuda_convnet

cuda_convnet.Conv2DCCLayer
cuda_convnet.MaxPool2DCCLayer
cuda_convnet.ShuffleBC01ToC01BLayer
cuda_convnet.bc01_to_c01b
cuda_convnet.ShuffleC01BToBC01Layer
cuda_convnet.c01b_to_bc01
cuda_convnet.NINLayer_c01b

lasagne.layers.dnn

dnn.Conv2DDNNLayer	2D convolutional layer
dnn.Conv3DDNNLayer	3D convolutional layer
dnn.MaxPool2DDNNLayer	2D max-pooling layer
dnn.Pool2DDNNLayer	2D pooling layer
dnn.MaxPool3DDNNLayer	3D max-pooling layer
dnn.Pool3DDNNLayer	3D pooling layer
dnn.SpatialPyramidPoolingDNNLayer	Spatial Pyramid Pooling Layer
dnn.BatchNormDNNLayer	Batch Normalization
dnn.batch_norm_dnn	Apply cuDNN batch normalization to an existing layer.

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
adamax	Adamax updates
amsgrad	AMSGrad 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.

Examples

Using nesterov_momentum() to define an update dictionary for a toy example network:

```
>>> 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 nesterov_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 = nesterov_momentum(loss, params, learning_rate=1e-4, momentum=.9)
>>> train_fn = theano.function([x, y], updates=updates)
```

With apply_momentum() and apply_nesterov_momentum(), we can add momentum to optimization schemes that do not usually support this:

```
>>> updates = lasagne.updates.rmsprop(loss, params, learning_rate=0.0001)
>>> updates = lasagne.updates.apply_momentum(updates, params, momentum=0.9)
```

All optimization schemes support symbolic variables for their hyperparameters, such as shared variables. This allows to vary hyperparameters during training without recompiling the training function. Note that the dtypes must match

the dtypes of the network parameters, which follow Theano's `floatX` setting. In the following example, we use `lasagne.utils.floatX()` to ensure this:

```
>>> eta = theano.shared(lasagne.utils.floatX(0.001))
>>> updates = lasagne.updates.adam(loss, params, learning_rate=eta)
>>> train_fn = theano.function([x, y], updates=updates)
>>> # we can now modify the learning rate at any time during training:
>>> eta.set_value(lasagne.utils.floatX(eta.get_value() * 0.1))
```

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

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 [1] 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 [2].

References

[1], [2]

`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 [1] 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

[1]

lasagne.updates.**adadelta**(*loss_or_grads*, *params*, *learning_rate=1.0*, *rho=0.95*, *epsilon=1e-06*)
Adadelta updates

Scale learning rates by the ratio of accumulated gradients to accumulated updates, see [1] 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:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \eta \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{r_t + \epsilon}}$$
$$s_t = \rho s_{t-1} + (1-\rho) * (\eta_t * g)^2$$

References

[1]

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 [1].

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] Learning rate

beta1 [float or symbolic scalar] Exponential decay rate for the first moment estimates.

beta2 [float or symbolic scalar] Exponential decay rate for the second moment estimates.

epsilon [float or symbolic scalar] Constant for numerical stability.

Returns

OrderedDict A dictionary mapping each parameter to its update expression

Notes

The paper [1] 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

[1]

`lasagne.updates.`**`adamax`**(*loss_or_grads*, *params*, *learning_rate=0.002*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)

Adamax updates

Adamax updates implemented as in [1]. This is a variant of of the Adam algorithm based on the infinity norm.

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] Learning rate

beta1 [float or symbolic scalar] Exponential decay rate for the first moment estimates.

beta2 [float or symbolic scalar] Exponential decay rate for the weighted infinity norm estimates.

epsilon [float or symbolic scalar] Constant for numerical stability.

Returns

OrderedDict A dictionary mapping each parameter to its update expression

References

[1]

`lasagne.updates.`**`amsgrad`**(*loss_or_grads*, *params*, *learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*)

AMSGrad updates

AMSGrad updates implemented as in [1].

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] Learning rate

beta1 [float or symbolic scalar] Exponential decay rate for the first moment estimates.

beta2 [float or symbolic scalar] Exponential decay rate for the second moment estimates.

epsilon [float or symbolic scalar] Constant for numerical stability.

Returns

OrderedDict A dictionary mapping each parameter to its update expression

References

[1]

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.

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 [1].

Parameters

tensor_vars [List of TensorVariables.] Tensors to be rescaled.

max_norm [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

[1]

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)
```

lasagne.init

Functions to create initializers for parameter variables.

Examples

```
>>> from lasagne.layers import DenseLayer
>>> from lasagne.init import Constant, GlorotUniform
>>> l1 = DenseLayer((100,20), num_units=50,
...                 W=GlorotUniform('relu'), b=Constant(0.0))
```

Initializers

Constant([val])	Initialize weights with constant value.
Normal([std, mean])	Sample initial weights from the Gaussian distribution.
Uniform([range, std, mean])	Sample initial weights from the uniform distribution.
Glorot(initializer[, gain, c01b])	Glorot weight initialization.
GlorotNormal([gain, c01b])	Glorot with weights sampled from the Normal distribution.
GlorotUniform([gain, c01b])	Glorot with weights sampled from the Uniform distribution.
He(initializer[, gain, c01b])	He weight initialization.
HeNormal([gain, c01b])	He initializer with weights sampled from the Normal distribution.
HeUniform([gain, c01b])	He initializer with weights sampled from the Uniform distribution.
Orthogonal([gain])	Intialize weights as Orthogonal matrix.
Sparse([sparsity, std])	Initialize weights as sparse matrix.

Detailed description

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.

This is also known as Xavier initialization [1].

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, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. 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

$$a = \sqrt{\frac{12}{fan_{in} + fan_{out}}}$$
$$W \sim U[-a, a]$$

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

[1]

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.

Weights are initialized with a standard deviation of $\sigma = gain\sqrt{\frac{1}{fan_{in}}}$ [1].

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, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. 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

[1]

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 [1]. 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'] Scaling factor for the weights. Set this to `1.0` for linear and sigmoid units, to 'relu' or `sqrt(2)` for rectified linear units, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. Other transfer functions may need different factors.

References

[1]

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).

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
`tanh`(x)	Tanh activation function $\varphi(x) = \tanh(x)$
`ScaledTanH`([scale_in, scale_out])	Scaled tanh $\varphi(x) = \tanh(\alpha \cdot x) \cdot \beta$
`rectify`(x)	Rectify activation function $\varphi(x) = \max(0, x)$
`LeakyRectify`([leakiness])	Leaky rectifier $\varphi(x) = (x > 0)?x : \alpha \cdot x$
`leaky_rectify`(x)	Instance of `LeakyRectify` with leakiness $\alpha = 0.01$
`very_leaky_rectify`(x)	Instance of `LeakyRectify` with leakiness $\alpha = 1/3$
`elu`(x)	Exponential Linear Unit $\varphi(x) = (x > 0)?x : e^x - 1$
`SELU`([scale, scale_neg])	Scaled Exponential Linear Unit
`selu`(x)	Instance of `SELU` with :math:'alphaapprox 1.6733,
`softplus`(x)	Softplus activation function $\varphi(x) = \log(1 + e^x)$
`linear`(x)	Linear activation function $\varphi(x) = x$
`identity`(x)	Linear activation function $\varphi(x) = x$

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.

class lasagne.nonlinearities.**ScaledTanH**(*scale_in=1*, *scale_out=1*)
Scaled tanh $\varphi(x) = \tanh(\alpha \cdot x) \cdot \beta$

This is a modified tanh function which allows to rescale both the input and the output of the activation.

Scaling the input down will result in decreasing the maximum slope of the tanh and as a result it will be in the linear regime in a larger interval of the input space. Scaling the input up will increase the maximum slope of the tanh and thus bring it closer to a step function.

Scaling the output changes the output interval to $[-\beta, \beta]$.

Parameters

scale_in [float32] The scale parameter α for the input

scale_out [float32] The scale parameter β for the output

Notes

LeCun et al. (in [1], Section 4.4) suggest `scale_in=2./3` and `scale_out=1.7159`, which has $\varphi(\pm 1) = \pm 1$, maximum second derivative at 1, and an effective gain close to 1.

By carefully matching α and β, the nonlinearity can also be tuned to preserve the mean and variance of its input:

- `scale_in=0.5`, `scale_out=2.4`: If the input is a random normal variable, the output will have zero mean and unit variance.
- `scale_in=1`, `scale_out=1.6`: Same property, but with a smaller linear regime in input space.

- `scale_in=0.5`, `scale_out=2.27`: If the input is a uniform normal variable, the output will have zero mean and unit variance.
- `scale_in=1`, `scale_out=1.48`: Same property, but with a smaller linear regime in input space.

References

[1], [2]

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 ScaledTanH
>>> scaled_tanh = ScaledTanH(scale_in=0.5, scale_out=2.27)
>>> l1 = DenseLayer(l_in, num_units=200, nonlinearity=scaled_tanh)
```

Methods

__call__(x)	Apply the scaled tanh function to the activation *x*.

lasagne.nonlinearities.**ScaledTanh**
alias of ScaledTanH

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 $\varphi(x) = (x > 0)?x : \alpha \cdot x$

The leaky rectifier was introduced in [1]. 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 [1].

very_leaky_rectify Instance with high leakiness of 1/3, as in [2].

References

[1], [2]

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.**elu**(x)
Exponential Linear Unit $\varphi(x) = (x > 0)?x : e^x - 1$

The Exponential Linear Unit (ELU) was introduced in [1]. Compared to the linear rectifier rectify(), it has a mean activation closer to zero and nonzero gradient for negative input, which can help convergence. Compared to the leaky rectifier LeakyRectify, it saturates for highly negative inputs.

Parameters

x [float32] The activation (the summed, weighed input of a neuron).

Returns

float32 The output of the exponential linear unit for the activation.

Notes

In [1], an additional parameter α controls the (negative) saturation value for negative inputs, but is set to 1 for all experiments. It is omitted here.

References

[1]

class lasagne.nonlinearities.**SELU**(*scale=1*, *scale_neg=1*)
Scaled Exponential Linear Unit $\varphi(x) = \lambda\,[(x > 0)?x : \alpha(e^x - 1)]$

The Scaled Exponential Linear Unit (SELU) was introduced in [1] as an activation function that allows the construction of self-normalizing neural networks.

Parameters

scale [float32] The scale parameter λ for scaling all output.

scale_neg [float32] The scale parameter α for scaling output for nonpositive argument values.

See also:

selu Instance with $\alpha \approx 1.6733, \lambda \approx 1.0507$ as used in [1].

References

[1]

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 SELU
>>> selu = SELU(2, 3)
>>> l1 = DenseLayer(l_in, num_units=200, nonlinearity=selu)
```

Methods

__call__(x)	Apply the SELU function to the activation x.

lasagne.nonlinearities.**selu**(*x*)
Instance of SELU with $\alpha \approx 1.6733, \lambda \approx 1.0507$

This has a stable and attracting fixed point of $\mu = 0$, $\sigma = 1$ under the assumptions of the original paper on self-normalizing neural networks.

lasagne.nonlinearities.**softplus**(*x*)
Softplus activation function $\varphi(x) = \log(1 + e^x)$

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the softplus function applied to the activation.

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.

lasagne.nonlinearities.**identity**(*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.

lasagne.objectives

Provides some minimal help with building loss expressions for training or validating a neural network.

Six 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.
binary_hinge_loss	Computes the binary hinge loss between predictions and targets.
multiclass_hinge_loss	Computes the multi-class hinge loss between predictions and targets.
huber_loss	Computes the huber loss between predictions and targets.

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.

Finally, two functions compute evaluation measures that are useful for validation and testing only, not for training:

binary_accuracy	Computes the binary accuracy between predictions and targets.
categorical_accuracy	Computes the categorical accuracy between predictions and targets.

Those can also be aggregated into a scalar expression if needed.

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.

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.

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$

p are the predictions, t are the targets, i denotes the data point and j denotes the class.

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. In the case of an integer vector argument, each element represents the position of the '1' in a one-hot encoding.

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 element-wise squared difference.

Notes

This is the loss function of choice for many regression problems or auto-encoders with linear output units.

`lasagne.objectives.`**`binary_hinge_loss`**(*predictions*, *targets*, *delta=1*, *log_odds=None*, *binary=True*)

Computes the binary hinge loss between predictions and targets.

$$L_i = \max(0, \delta - t_i p_i)$$

Parameters

predictions [Theano tensor] Predictions in (0, 1), such as sigmoidal output of a neural network (or log-odds of predictions depending on *log_odds*).

targets [Theano tensor] Targets in {0, 1} (or in {-1, 1} depending on *binary*), such as ground truth labels.

delta [scalar, default 1] The hinge loss margin

log_odds [bool, default None] `False` if predictions are sigmoid outputs in (0, 1), `True` if predictions are sigmoid inputs, or log-odds. If `None`, will assume `True`, but warn that the default will change to `False`.

binary [bool, default True] `True` if targets are in {0, 1}, `False` if they are in {-1, 1}

Returns

Theano tensor An expression for the element-wise binary hinge loss

Notes

This is an alternative to the binary cross-entropy loss for binary classification problems.

Note that it is a drop-in replacement only when giving `log_odds=False`. Otherwise, it requires log-odds rather than sigmoid outputs. Be aware that depending on the Theano version, `log_odds=False` with a sigmoid output layer may be less stable than `log_odds=True` with a linear layer.

`lasagne.objectives.`**`multiclass_hinge_loss`**(*predictions*, *targets*, *delta=1*)

Computes the multi-class hinge loss between predictions and targets.

$$L_i = \max_{j \neq t_i}(0, p_j - p_{t_i} + \delta)$$

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 a vector of int giving the correct class index per data point or a 2D tensor of one-hot encoding of the correct class in the same layout as predictions (non-binary targets in [0, 1] do not work!)

delta [scalar, default 1] The hinge loss margin

Returns

Theano 1D tensor An expression for the item-wise multi-class hinge loss

Notes

This is an alternative to the categorical cross-entropy loss for multi-class classification problems

`lasagne.objectives.`**`huber_loss`**(*predictions*, *targets*, *delta=1*)

Computes the huber loss between predictions and targets.

$$L_i = \frac{(p - t)^2}{2}, |p - t| \leq \delta$$
$$L_i = \delta(|p - t| - \frac{\delta}{2}), |p - t|\delta$$

Parameters

predictions [Theano 2D tensor or 1D tensor] Prediction outputs of a neural network.

targets [Theano 2D tensor or 1D tensor] Ground truth to which the prediction is to be compared with. Either a vector or 2D Tensor.

delta [scalar, default 1] This delta value is defaulted to 1, for *SmoothL1Loss* described in Fast-RCNN paper [1] .

Returns

Theano tensor An expression for the element-wise huber loss [2] .

Notes

This is an alternative to the squared error for regression problems.

References

[1], [2]

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 data point contributes more to the loss when it shares a batch with low-weighted or masked data points than with high-weighted ones.

Evaluation functions

lasagne.objectives.**binary_accuracy**(*predictions*, *targets*, *threshold=0.5*)

Computes the binary accuracy between predictions and targets.

$$L_i = I(t_i = I(p_i \geq \alpha))$$

Parameters

predictions [Theano tensor] Predictions in [0, 1], such as a sigmoidal output of a neural network, giving the probability of the positive class

targets [Theano tensor] Targets in {0, 1}, such as ground truth labels.

threshold [scalar, default: 0.5] Specifies at what threshold to consider the predictions being of the positive class

Returns

Theano tensor An expression for the element-wise binary accuracy in {0, 1}

Notes

This objective function should not be used with a gradient calculation; its gradient is zero everywhere. It is intended as a convenience for validation and testing, not training.

To obtain the average accuracy, call `theano.tensor.mean()` on the result, passing `dtype=theano.config.floatX` to compute the mean on GPU.

`lasagne.objectives.`**`categorical_accuracy`**(*predictions*, *targets*, *top_k=1*)

Computes the categorical accuracy between predictions and targets.

$$L_i = I(t_i = \operatorname{argmax}_c p_{i,c})$$

Can be relaxed to allow matches among the top k predictions:

$$L_i = I(t_i \in \operatorname{argsort}_c(-p_{i,c})_{:k})$$

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 a vector of int giving the correct class index per data point or a 2D tensor of 1 hot encoding of the correct class in the same layout as predictions

top_k [int] Regard a prediction to be correct if the target class is among the *top_k* largest class probabilities. For the default value of 1, a prediction is correct only if the target class is the most probable.

Returns

Theano 1D tensor An expression for the item-wise categorical accuracy in {0, 1}

Notes

This is a strictly non differential function as it includes an argmax. This objective function should never be used with a gradient calculation. It is intended as a convenience for validation and testing not training.

To obtain the average accuracy, call `theano.tensor.mean()` on the result, passing `dtype=theano.config.floatX` to compute the mean on GPU.

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

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
```

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

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)

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.

Note: When using cuDNN, the backward passes of convolutional and max-pooling layers will introduce additional nondeterminism (for performance reasons). For 2D convolutions, you can enforce a deterministic backward pass implementation via the Theano flags `dnn.conv.algo_bwd_filter=deterministic` and `dnn.conv.algo_bwd_data=deterministic`. Alternatively, you can disable cuDNN completely with `dnn.enabled=False`.

`lasagne.random.`**`get_rng`**`()`
Get the package-level random number generator.

Returns

:class:'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.

lasagne.utils

`lasagne.utils.`**`int_types = (numbers.Integral, np.integer)`**
Tuple of `int`-like types for `isinstance` checks. Specifically includes long integers and numpy integers.

`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.**collect_shared_vars**(*expressions*)
Returns all shared variables the given expression(s) depend on.

Parameters

expressions [Theano expression or iterable of Theano expressions] The expressions to collect shared variables from.

Returns

list of Theano shared variables All shared variables the given expression(s) depend on, in fixed order (as found by a left-recursive depth-first search). If some expressions are shared variables themselves, they are included in the result.

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 [numpy array or Theano expression] Weight or bias.

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. Finally, in case *array* is a vector, *norm_axes* is set to an empty tuple, and this function will simply return the absolute value for each element. This is useful when the function is applied to all parameters of the network, including the bias, without distinction. (Optional)

Returns

norms [1D array or Theano vector (1D)] 1D array or Theano vector of incoming weight/bias 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 [scalar number, numpy array, Theano expression, or callable] Either of the following:

- a scalar or a numpy array with the initial parameter values
- a Theano expression or 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, a Theano expression, or a shared variable representing the parameters.

shape [iterable of int] a tuple or other iterable of integers representing the desired shape of the parameter array.

name [string, optional] The name to give to the parameter variable. Ignored if *spec* is or returns a Theano expression or shared variable that already has a name.

Returns

Theano shared variable or Theano expression A Theano shared variable or expression representing layer parameters. If a scalar or a numpy array was provided, a shared variable is initialized to contain this array. If a shared variable or expression was provided, it is simply

returned. If a callable was provided, it is called, and its output is used to initialize a shared 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 scalars, numpy arrays, existing Theano shared variables or expressions, and callables for generating initial parameter values, Theano expressions, or shared variables.

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

[1] Lin, Min, Qiang Chen, and Shuicheng Yan (2013): Network in network. arXiv preprint arXiv:1312.4400.

[1] Vincent Dumoulin, Francesco Visin (2016): A guide to convolution arithmetic for deep learning. arXiv. http://arxiv.org/abs/1603.07285, https://github.com/vdumoulin/conv_arithmetic

[1] Fisher Yu, Vladlen Koltun (2016), Multi-Scale Context Aggregation by Dilated Convolutions. ICLR 2016. http://arxiv.org/abs/1511.07122, https://github.com/fyu/dilation

[1] He, Kaiming et al (2015): Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. http://arxiv.org/pdf/1406.4729.pdf.

[1] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[1] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[1] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[1] Cho, Kyunghyun, et al: On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259 (2014).

[2] Chung, Junyoung, et al.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555 (2014).

[3] Graves, Alex: "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[1] Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." Neural computation 12.10 (2000): 2451-2471.

[1] 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.

[2] 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.

[1] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, C. Bregler (2014): Efficient Object Localization Using Convolutional Networks. https://arxiv.org/abs/1411.4280

[1] 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.

[1] Ioffe, Sergey and Szegedy, Christian (2015): Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. http://arxiv.org/abs/1502.03167.

[1] Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016): Instance Normalization: The Missing Ingredient for Fast Stylization. https://arxiv.org/abs/1607.08022.

[2] Ba, J., Kiros, J., & Hinton, G. (2016): Layer normalization. https://arxiv.org/abs/1607.06450.

[1] Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016): Instance Normalization: The Missing Ingredient for Fast Stylization. https://arxiv.org/abs/1607.08022.

[1] Ba, J., Kiros, J., & Hinton, G. (2016): Layer normalization. https://arxiv.org/abs/1607.06450.

[1] Max Jaderberg, Karen Simonyan, Andrew Zisserman, Koray Kavukcuoglu (2015): Spatial Transformer Networks. NIPS 2015, http://papers.nips.cc/paper/5854-spatial-transformer-networks.pdf

[1] Max Jaderberg, Karen Simonyan, Andrew Zisserman, Koray Kavukcuoglu (2015): Spatial Transformer Networks. NIPS 2015, http://papers.nips.cc/paper/5854-spatial-transformer-networks.pdf

[2] Fred L. Bookstein (1989): Principal warps: thin-plate splines and the decomposition of deformations. IEEE Transactions on Pattern Analysis and Machine Intelligence. http://doi.org/10.1109/34.24792

[1] K He, X Zhang et al. (2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, http://arxiv.org/abs/1502.01852

[1] Bing Xu, Naiyan Wang et al. (2015): Empirical Evaluation of Rectified Activations in Convolutional Network, http://arxiv.org/abs/1505.00853

[1] He, Kaiming et al (2015): Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. http://arxiv.org/pdf/1406.4729.pdf.

[1] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12:2121-2159.

[2] Chris Dyer: Notes on AdaGrad. http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf

[1] 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)

[1] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.

[1] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[1] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[1] https://openreview.net/forum?id=ryQu7f-RZ

[1] 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).

[1] Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

[1] Kaiming He et al. (2015): Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. arXiv preprint arXiv:1502.01852.

[1] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." arXiv preprint arXiv:1312.6120 (2013).

[1] LeCun, Yann A., et al. (1998): Efficient BackProp, http://link.springer.com/chapter/10.1007/3-540-49430-8_2, http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[2] Masci, Jonathan, et al. (2011): Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction, http://link.springer.com/chapter/10.1007/978-3-642-21735-7_7, http://people.idsia.ch/~ciresan/data/icann2011.pdf

[1] Maas et al. (2013): Rectifier Nonlinearities Improve Neural Network Acoustic Models, http://web.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf

[2] Graham, Benjamin (2014): Spatially-sparse convolutional neural networks, http://arxiv.org/abs/1409.6070

[1] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter (2015): Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), http://arxiv.org/abs/1511.07289

[1] Günter Klambauer et al. (2017): Self-Normalizing Neural Networks, https://arxiv.org/abs/1706.02515

[1] Ross Girshick et al (2015): Fast RCNN https://arxiv.org/pdf/1504.08083.pdf

[2] Huber, Peter et al (1964) Robust Estimation of a Location Parameter https://projecteuclid.org/euclid.aoms/1177703732

l

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V