Contents

1 LICENSE 1

2 Deep Learning Tutorials 3

3 Getting Started 5
   3.1 Download .................................................. 5
   3.2 Datasets .................................................. 5
   3.3 Notation ................................................. 7
   3.4 A Primer on Supervised Optimization for Deep Learning ............... 9
   3.5 Theano/Python Tips .................................... 15

4 Classifying MNIST digits using Logistic Regression 17
   4.1 The Model .................................................. 17
   4.2 Defining a Loss Function ................................ 19
   4.3 Creating a LogisticRegression class .................... 19
   4.4 Learning the Model ..................................... 22
   4.5 Testing the model ....................................... 23
   4.6 Putting it All Together .................................. 25
   4.7 Prediction Using a Trained Model ....................... 34

5 Multilayer Perceptron 37
   5.1 The Model .................................................. 37
   5.2 Going from logistic regression to MLP .................... 38
   5.3 Putting it All Together .................................. 43
   5.4 Tips and Tricks for training MLPs ....................... 51

6 Convolutional Neural Networks (LeNet) 53
   6.1 Motivation .................................................. 53
   6.2 Sparse Connectivity ..................................... 54
   6.3 Shared Weights .......................................... 54
   6.4 Details and Notation ..................................... 55
   6.5 The Convolution Operator ................................ 56
   6.6 MaxPooling ............................................... 58
13.1 Summary ................................................................. 147
13.2 Data .................................................................. 147
13.3 Model .................................................................. 148
13.4 Code - Citations - Contact .............................................. 151
13.5 References ............................................................... 152

14 Modeling and generating sequences of polyphonic music with the RNN-RBM 153
14.1 The RNN-RBM ......................................................... 153
14.2 Implementation ....................................................... 154
14.3 Results .................................................................. 160
14.4 How to improve this code .............................................. 162

15 Miscellaneous ............................................................. 163
15.1 Plotting Samples and Filters ...................................... 163

16 References ................................................................. 167

Bibliography ...................................................................... 169
Copyright (c) 2008–2013, Theano Development Team All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Theano nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS \"AS IS\" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals: Artificial Intelligence. See these course notes for a brief introduction to Machine Learning for AI and an introduction to Deep Learning algorithms.

Deep Learning is about learning multiple levels of representation and abstraction that help to make sense of data such as images, sound, and text. For more about deep learning algorithms, see for example:

- The monograph or review paper Learning Deep Architectures for AI (Foundations & Trends in Machine Learning, 2009).
- The ICML 2009 Workshop on Learning Feature Hierarchies webpage has a list of references.
- The LISA public wiki has a reading list and a bibliography.
- Geoff Hinton has readings from 2009’s NIPS tutorial.

The tutorials presented here will introduce you to some of the most important deep learning algorithms and will also show you how to run them using Theano. Theano is a python library that makes writing deep learning models easy, and gives the option of training them on a GPU.

The algorithm tutorials have some prerequisites. You should know some python, and be familiar with numpy. Since this tutorial is about using Theano, you should read over the Theano basic tutorial first. Once you’ve done that, read through our Getting Started chapter – it introduces the notation, and downloadable datasets used in the algorithm tutorials, and the way we do optimization by stochastic gradient descent.

The code is available on the Deep Learning Tutorial repositories.

The purely supervised learning algorithms are meant to be read in order:

1. *Logistic Regression* - using Theano for something simple
2. *Multilayer perceptron* - introduction to layers
3. *Deep Convolutional Network* - a simplified version of LeNet5
The unsupervised and semi-supervised learning algorithms can be read in any order (the auto-encoders can be read independently of the RBM/DBN thread):

- **Auto Encoders, Denoising Autoencoders** - description of autoencoders
- **Stacked Denoising Auto-Encoders** - easy steps into unsupervised pre-training for deep nets
- **Restricted Boltzmann Machines** - single layer generative RBM model
- **Deep Belief Networks** - unsupervised generative pre-training of stacked RBMs followed by supervised fine-tuning

Building towards including the mcRBM model, we have a new tutorial on sampling from energy models:

- **HMC Sampling** - hybrid (aka Hamiltonian) Monte-Carlo sampling with scan()

Building towards including the Contractive auto-encoders tutorial, we have the code for now:

- **Contractive auto-encoders code** - There is some basic doc in the code.

**Recurrent neural networks with word embeddings and context window:**

- **Semantic Parsing of Speech using Recurrent Net**

**LSTM network for sentiment analysis:**

- **LSTM network**

**Energy-based recurrent neural network (RNN-RBM):**

- **Modeling and generating sequences of polyphonic music**

Note that the tutorials here are all compatible with Python 2 and 3, with the exception of *Modeling and generating sequences of polyphonic music with the RNN-RBM* which is only available for Python 2.
These tutorials do not attempt to make up for a graduate or undergraduate course in machine learning, but we do make a rapid overview of some important concepts (and notation) to make sure that we’re on the same page. You’ll also need to download the datasets mentioned in this chapter in order to run the example code of the upcoming tutorials.

3.1 Download

On each learning algorithm page, you will be able to download the corresponding files. If you want to download all of them at the same time, you can clone the git repository of the tutorial:

```
```

On Linux or Mac systems, after cloning, all datasets can be downloaded at once with:

```
cd DeepLearningTutorials/data .download.sh
```

3.2 Datasets

3.2.1 MNIST Dataset

(mnist.pkl.gz)

The MNIST dataset consists of handwritten digit images and it is divided in 60,000 examples for the training set and 10,000 examples for testing. In many papers as well as in this tutorial, the official training set of 60,000 is divided into an actual training set of 50,000 examples and 10,000 validation examples (for selecting hyper-parameters like learning rate and size of the
model). All digit images have been size-normalized and centered in a fixed size image of 28 x 28 pixels. In the original dataset each pixel of the image is represented by a value between 0 and 255, where 0 is black, 255 is white and anything in between is a different shade of grey.

Here are some examples of MNIST digits:

504192

For convenience we pickled the dataset to make it easier to use in python. It is available for download here. The pickled file represents a tuple of 3 lists: the training set, the validation set and the testing set. Each of the three lists is a pair formed from a list of images and a list of class labels for each of the images. An image is represented as numpy 1-dimensional array of 784 (28 x 28) float values between 0 and 1 (0 stands for black, 1 for white). The labels are numbers between 0 and 9 indicating which digit the image represents. The code block below shows how to load the dataset.

```python
import cPickle, gzip, numpy

# Load the dataset
f = gzip.open('mnist.pkl.gz', 'rb')
train_set, valid_set, test_set = cPickle.load(f)
f.close()
```

When using the dataset, we usually divide it in minibatches (see Stochastic Gradient Descent). We encourage you to store the dataset into shared variables and access it based on the minibatch index, given a fixed and known batch size. The reason behind shared variables is related to using the GPU. There is a large overhead when copying data into the GPU memory. If you would copy data on request (each minibatch individually when needed) as the code will do if you do not use shared variables, due to this overhead, the GPU code will not be much faster then the CPU code (maybe even slower). If you have your data in Theano shared variables though, you give Theano the possibility to copy the entire data on the GPU in a single call when the shared variables are constructed. Afterwards the GPU can access any minibatch by taking a slice from this shared variables, without needing to copy any information from the CPU memory and therefore bypassing the overhead. Because the datapoints and their labels are usually of different nature (labels are usually integers while datapoints are real numbers) we suggest to use different variables for label and data. Also we recommend using different variables for the training set, validation set and testing set to make the code more readable (resulting in 6 different shared variables).

Since now the data is in one variable, and a minibatch is defined as a slice of that variable, it comes more natural to define a minibatch by indicating its index and its size. In our setup the batch size stays constant throughout the execution of the code, therefore a function will actually require only the index to identify on which datapoints to work. The code below shows how to store your data and how to access a minibatch:

```python
def shared_dataset(data_xy):
    """Function that loads the dataset into shared variables

    The reason we store our dataset in shared variables is to allow Theano to copy it into the GPU memory (when code is run on GPU).
    """
```
Since copying data into the GPU is slow, copying a minibatch of data_x, data_y = data_xy
shared_x = theano.shared(numpy.asarray(data_x, dtype=theano.config.floatX))
shared_y = theano.shared(numpy.asarray(data_y, dtype=theano.config.floatX))
# When storing data on the GPU it has to be stored as floats
# therefore we will store the labels as `floatX` as well
# (`shared_y` does exactly that). But during our computations
# we need them as ints (we use labels as index, and if they are
# floats it doesn't make sense) therefore instead of returning
# `shared_y` we will have to cast it to int. This little hack
# lets us get around this issue
return shared_x, T.cast(shared_y, 'int32')

test_set_x, test_set_y = shared_dataset(test_set)
valid_set_x, valid_set_y = shared_dataset(valid_set)
train_set_x, train_set_y = shared_dataset(train_set)

batch_size = 500  # size of the minibatch

# accessing the third minibatch of the training set

Data has to be stored as floats on the GPU (the right dtype for storing on the GPU is given by theano.config.floatX). To get around this shortcoming for the labels, we store them as float, and then cast it to int.

Note: If you are running your code on the GPU and the dataset you are using is too large to fit in memory the code will crash. In such a case you should store the data in a shared variable. You can however store a sufficiently small chunk of your data (several minibatches) in a shared variable and use that during training. Once you got through the chunk, update the values it stores. This way you minimize the number of data transfers between CPU memory and GPU memory.

3.3 Notation

3.3.1 Dataset notation

We label data sets as $D$. When the distinction is important, we indicate train, validation, and test sets as: $D_{train}$, $D_{valid}$, and $D_{test}$. The validation set is used to perform model selection and hyper-parameter selection, whereas the test set is used to evaluate the final generalization error and compare different algorithms.
in an unbiased way.

The tutorials mostly deal with classification problems, where each data set \( D \) is an indexed set of pairs \((x^{(i)}, y^{(i)})\). We use superscripts to distinguish training set examples: \( x^{(i)} \in \mathbb{R}^D \) is thus the \( i \)-th training example of dimensionality \( D \). Similarly, \( y^{(i)} \in \{0, ..., L\} \) is the \( i \)-th label assigned to input \( x^{(i)} \). It is straightforward to extend these examples to ones where \( y^{(i)} \) has other types (e.g. Gaussian for regression, or groups of multinomials for predicting multiple symbols).

### 3.3.2 Math Conventions

- \( W \): upper-case symbols refer to a matrix unless specified otherwise
- \( W_{ij} \): element at \( i \)-th row and \( j \)-th column of matrix \( W \)
- \( W_i, W_i^\top \): vector, \( i \)-th row of matrix \( W \)
- \( W_{-j} \): vector, \( j \)-th column of matrix \( W \)
- \( b \): lower-case symbols refer to a vector unless specified otherwise
- \( b_i \): \( i \)-th element of vector \( b \)

### 3.3.3 List of Symbols and acronyms

- \( D \): number of input dimensions.
- \( D_h^{(i)} \): number of hidden units in the \( i \)-th layer.
- \( f_\theta(x), f(x) \): classification function associated with a model \( P(Y|x, \theta) \), defined as \( \arg\max_k P(Y = k|x, \theta) \). Note that we will often drop the \( \theta \) subscript.
- \( L \): number of labels.
- \( \mathcal{L}(\theta, D) \): log-likelihood \( D \) of the model defined by parameters \( \theta \).
- \( \ell(\theta, D) \) empirical loss of the prediction function \( f \) parameterized by \( \theta \) on data set \( D \).
- NLL: negative log-likelihood
- \( \theta \): set of all parameters for a given model

### 3.3.4 Python Namespaces

Tutorial code often uses the following namespaces:

```python
import theano
import theano.tensor as T
import numpy
```
3.4 A Primer on Supervised Optimization for Deep Learning

What’s exciting about Deep Learning is largely the use of unsupervised learning of deep networks. But supervised learning also plays an important role. The utility of unsupervised pre-training is often evaluated on the basis of what performance can be achieved after supervised fine-tuning. This chapter reviews the basics of supervised learning for classification models, and covers the minibatch stochastic gradient descent algorithm that is used to fine-tune many of the models in the Deep Learning Tutorials. Have a look at these introductory course notes on gradient-based learning for more basics on the notion of optimizing a training criterion using the gradient.

3.4.1 Learning a Classifier

Zero-One Loss

The models presented in these deep learning tutorials are mostly used for classification. The objective in training a classifier is to minimize the number of errors (zero-one loss) on unseen examples. If \( f : R^D \rightarrow \{0, ..., L\} \) is the prediction function, then this loss can be written as:

\[
\ell_{0,1} = \sum_{i=0}^{\vert D \vert} I_{f(x^{(i)}) \neq y^{(i)}}
\]

where either \( D \) is the training set (during training) or \( D \cap D_{\text{train}} = \emptyset \) (to avoid biasing the evaluation of validation or test error). \( I \) is the indicator function defined as:

\[
I_x = \begin{cases} 
1 & \text{if } x \text{ is True} \\
0 & \text{otherwise} 
\end{cases}
\]

In this tutorial, \( f \) is defined as:

\[
f(x) = \arg\max_k P(Y = k|x, \theta)
\]

In python, using Theano this can be written as:

```python
# zero_one_loss is a Theano variable representing a symbolic
# expression of the zero one loss ; to get the actual value this
# symbolic expression has to be compiled into a Theano function (see
# the Theano tutorial for more details)
zero_one_loss = T.sum(T.neq(T.argmax(p_y_given_x), y))
```

Negative Log-Likelihood Loss

Since the zero-one loss is not differentiable, optimizing it for large models (thousands or millions of parameters) is prohibitively expensive (computationally). We thus maximize the log-likelihood of our classifier given all the labels in a training set.

\[
\mathcal{L}(\theta, D) = \sum_{i=0}^{\vert D \vert} \log P(Y = y^{(i)}|x^{(i)}, \theta)
\]
The likelihood of the correct class is not the same as the number of right predictions, but from the point of view of a randomly initialized classifier they are pretty similar. Remember that likelihood and zero-one loss are different objectives; you should see that they are correlated on the validation set but sometimes one will rise while the other falls, or vice-versa.

Since we usually speak in terms of minimizing a loss function, learning will thus attempt to **minimize** the **negative** log-likelihood (NLL), defined as:

\[
NLL(\theta, D) = -\sum_{i=0}^{|D|} \log P(Y = y^{(i)} | x^{(i)}, \theta)
\]

The NLL of our classifier is a differentiable surrogate for the zero-one loss, and we use the gradient of this function over our training data as a supervised learning signal for deep learning of a classifier.

This can be computed using the following line of code:

```python
# NLL is a symbolic variable ; to get the actual value of NLL, this symbolic expression has to be compiled into a Theano function (see the Theano tutorial for more details)
NLL = -T.sum(T.log(p_y_given_x)[T.arange(y.shape[0]), y])
# note on syntax: T.arange(y.shape[0]) is a vector of integers [0,1,2,...,len(y)].
# Indexing a matrix M by the two vectors [0,1,...,K], [a,b,...,k] returns the elements M[0,a], M[1,b], ..., M[K,k] as a vector. Here, we use this syntax to retrieve the log-probability of the correct labels, y.
```

### 3.4.2 Stochastic Gradient Descent

What is ordinary gradient descent? it is a simple algorithm in which we repeatedly make small steps downward on an error surface defined by a loss function of some parameters. For the purpose of ordinary gradient descent we consider that the training data is rolled into the loss function. Then the pseudocode of this algorithm can be described as:

```python
# GRADIENT DESCENT
while True:
    loss = f(params)
    d_loss_wrt_params = ... # compute gradient
    params -= learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```

Stochastic gradient descent (SGD) works according to the same principles as ordinary gradient descent, but proceeds more quickly by estimating the gradient from just a few examples at a time instead of the entire training set. In its purest form, we estimate the gradient from just a single example at a time.

```python
# STOCHASTIC GRADIENT DESCENT
for (x_i, y_i) in training_set:
    # imagine an infinite generator
    # that may repeat examples (if there is only a
    # finite training set)
```
The variant that we recommend for deep learning is a further twist on stochastic gradient descent using so-called “minibatches”. Minibatch SGD (MSGD) works identically to SGD, except that we use more than one training example to make each estimate of the gradient. This technique reduces variance in the estimate of the gradient, and often makes better use of the hierarchical memory organization in modern computers.

```python
for (x_batch, y_batch) in train_batches:
    # imagine an infinite generator
    # that may repeat examples
    loss = f(params, x_batch, y_batch)
    d_loss_wrt_params = ... # compute gradient using theano
    params = params - learning_rate * d_loss_wrt_params
if <stopping condition is met>:
    return params
```

There is a tradeoff in the choice of the minibatch size \( B \). The reduction of variance and use of SIMD instructions helps most when increasing \( B \) from 1 to 2, but the marginal improvement fades rapidly to nothing. With large \( B \), time is wasted in reducing the variance of the gradient estimator, that time would be better spent on additional gradient steps. An optimal \( B \) is model-, dataset-, and hardware-dependent, and can be anywhere from 1 to maybe several hundreds. In the tutorial we set it to 20, but this choice is almost arbitrary (though harmless).

**Note:** If you are training for a fixed number of epochs, the minibatch size becomes important because it controls the number of updates done to your parameters. Training the same model for 10 epochs using a batch size of 1 yields completely different results compared to training for the same 10 epochs but with a batchsize of 20. Keep this in mind when switching between batch sizes and be prepared to tweak all the other parameters according to the batch size used.

All code-blocks above show pseudocode of how the algorithm looks like. Implementing such algorithm in Theano can be done as follows:

```python
# Minibatch Stochastic Gradient Descent

# assume loss is a symbolic description of the loss function given
# the symbolic variables params (shared variable), x_batch, y_batch;

# compute gradient of loss with respect to params
d_loss_wrt_params = T.grad(loss, params)

# compile the MSGD step into a theano function
updates = [(params, params - learning_rate * d_loss_wrt_params)]
MSGD = theano.function([x_batch, y_batch], loss, updates=updates)

for (x_batch, y_batch) in train_batches:
```
# here x_batch and y_batch are elements of train_batches and # therefore numpy arrays; function MSGD also updates the params

```python
print('Current loss is ', MSGD(x_batch, y_batch))
if stopping_condition_is_met:
    return params
```

## 3.4.3 Regularization

There is more to machine learning than optimization. When we train our model from data we are trying to prepare it to do well on new examples, not the ones it has already seen. The training loop above for MSGD does not take this into account, and may overfit the training examples. A way to combat overfitting is through regularization. There are several techniques for regularization; the ones we will explain here are L1/L2 regularization and early-stopping.

### L1 and L2 regularization

L1 and L2 regularization involve adding an extra term to the loss function, which penalizes certain parameter configurations. Formally, if our loss function is:

\[
NLL(\theta, D) = -\sum_{i=0}^{|D|} \log P(Y = y^{(i)} | x^{(i)}, \theta)
\]

then the regularized loss will be:

\[
E(\theta, D) = NLL(\theta, D) + \lambda R(\theta)
\]

or, in our case

\[
E(\theta, D) = NLL(\theta, D) + \lambda ||\theta||_p^p
\]

where

\[
||\theta||_p = \left( \sum_{j=0}^{\frac{||\theta||}{p}} |\theta_j|^p \right)^{\frac{1}{p}}
\]

which is the \(L_p\) norm of \(\theta\). \(\lambda\) is a hyper-parameter which controls the relative importance of the regularization parameter. Commonly used values for \(p\) are 1 and 2, hence the L1/L2 nomenclature. If \(p=2\), then the regularizer is also called “weight decay”.

In principle, adding a regularization term to the loss will encourage smooth network mappings in a neural network (by penalizing large values of the parameters, which decreases the amount of nonlinearity that the network models). More intuitively, the two terms (NLL and \(R(\theta)\)) correspond to modelling the data well (NLL) and having “simple” or “smooth” solutions (\(R(\theta)\)). Thus, minimizing the sum of both will, in theory, correspond to finding the right trade-off between the fit to the training data and the “generality” of the solution that is found. To follow Occam’s razor principle, this minimization should find us the simplest solution (as measured by our simplicity criterion) that fits the training data.
Note that the fact that a solution is “simple” does not mean that it will generalize well. Empirically, it was found that performing such regularization in the context of neural networks helps with generalization, especially on small datasets. The code block below shows how to compute the loss in python when it contains both a L1 regularization term weighted by $\lambda_1$ and L2 regularization term weighted by $\lambda_2$.

```python
# symbolic Theano variable that represents the L1 regularization term
L1 = T.sum(abs(param))

# symbolic Theano variable that represents the squared L2 term
L2 = T.sum(param ** 2)

# the loss
loss = NLL + lambda_1 * L1 + lambda_2 * L2
```

### Early-Stopping

Early-stopping combats overfitting by monitoring the model’s performance on a validation set. A validation set is a set of examples that we never use for gradient descent, but which is also not a part of the test set. The validation examples are considered to be representative of future test examples. We can use them during training because they are not part of the test set. If the model’s performance ceases to improve sufficiently on the validation set, or even degrades with further optimization, then the heuristic implemented here gives up on much further optimization.

The choice of when to stop is a judgement call and a few heuristics exist, but these tutorials will make use of a strategy based on a geometrically increasing amount of patience.

```python
# early-stopping parameters
patience = 5000  # look as this many examples regardless
patience_increase = 2  # wait this much longer when a new best is found
improvement_threshold = 0.995  # a relative improvement of this much is considered significant
validation_frequency = min(n_train_batches, patience/2)  # go through this many minibatches before checking the network
# on the validation set; in this case we check every epoch

best_params = None
best_validation_loss = numpy.inf
test_score = 0.
start_time = time.clock()
done_looping = False
epoch = 0

while (epoch < n_epochs) and (not done_looping):
    # Report "1" for first epoch, "n_epochs" for last epoch
    epoch = epoch + 1
    for minibatch_index in range(n_train_batches):
        d_loss_wrt_params = ...  # compute gradient
```
params -= learning_rate * d_loss_wrt_params  # gradient descent

# iteration number. We want it to start at 0.
iter = (epoch - 1) * n_train_batches + minibatch_index
# note that if we do `iter % validation_frequency` it will be
# true for iter = 0 which we do not want. We want it true for
# iter = validation_frequency - 1.
if (iter + 1) % validation_frequency == 0:
    this_validation_loss = ...  # compute zero-one loss on validation
    if this_validation_loss < best_validation_loss:
        # improve patience if loss improvement is good enough
        if this_validation_loss < best_validation_loss + improvement_threshold:
            patience = max(patience, iter + patience_increase)
            best_params = copy.deepcopy(params)
            best_validation_loss = this_validation_loss

        if patience <= iter:
            done_looping = True
            break

# POSTCONDITION:
# best_params refers to the best out-of-sample parameters observed during the optimization

If we run out of batches of training data before running out of patience, then we just go back to the beginning of the training set and repeat.

**Note:** The validation_frequency should always be smaller than the patience. The code should check at least two times how it performs before running out of patience. This is the reason we used the formulation validation_frequency = min( value, patience/2.)

**Note:** This algorithm could possibly be improved by using a test of statistical significance rather than the simple comparison, when deciding whether to increase the patience.

### 3.4.4 Testing

After the loop exits, the best_params variable refers to the best-performing model on the validation set. If we repeat this procedure for another model class, or even another random initialization, we should use the same train/valid/test split of the data, and get other best-performing models. If we have to choose what the best model class or the best initialization was, we compare the best_validation_loss for each model. When
we have finally chosen the model we think is the best (on validation data), we report that model’s test set performance. That is the performance we expect on unseen examples.

### 3.4.5 Recap

That’s it for the optimization section. The technique of early-stopping requires us to partition the set of examples into three sets (training $\mathcal{D}_{\text{train}}$, validation $\mathcal{D}_{\text{valid}}$, test $\mathcal{D}_{\text{test}}$). The training set is used for minibatch stochastic gradient descent on the differentiable approximation of the objective function. As we perform this gradient descent, we periodically consult the validation set to see how our model is doing on the real objective function (or at least our empirical estimate of it). When we see a good model on the validation set, we save it. When it has been a long time since seeing a good model, we abandon our search and return the best parameters found, for evaluation on the test set.

### 3.5 Theano/Python Tips

#### 3.5.1 Loading and Saving Models

When you’re doing experiments, it can take hours (sometimes days!) for gradient-descent to find the best parameters. You will want to save those weights once you find them. You may also want to save your current-best estimates as the search progresses.

**Pickle the numpy ndarrays from your shared variables**

The best way to save/archive your model’s parameters is to use pickle or deepcopy the ndarray objects. So for example, if your parameters are in shared variables $w$, $v$, $u$, then your save command should look something like:

```python
>>> import cPickle
>>> save_file = open('path', 'wb')  # this will overwrite current contents
>>> cPickle.dump(w.get_value(borrow=True), save_file, -1)  # the -1 is for
   \rightarrow HIGHEST_PROTOCOL
>>> cPickle.dump(v.get_value(borrow=True), save_file, -1)  # .. and it
   \rightarrow triggers much more efficient
>>> cPickle.dump(u.get_value(borrow=True), save_file, -1)  # .. storage than
   \rightarrow numpy's default
>>> save_file.close()
```

Then later, you can load your data back like this:

```python
>>> save_file = open('path')
>>> w.set_value(cPickle.load(save_file), borrow=True)
>>> v.set_value(cPickle.load(save_file), borrow=True)
>>> u.set_value(cPickle.load(save_file), borrow=True)
```

This technique is a bit verbose, but it is tried and true. You will be able to load your data and render it in matplotlib without trouble, years after saving it.

**Do not pickle your training or test functions for long-term storage**
Theano functions are compatible with Python’s deepcopy and pickle mechanisms, but you should not necessarily pickle a Theano function. If you update your Theano folder and one of the internal changes, then you may not be able to un-pickle your model. Theano is still in active development, and the internal APIs are subject to change. So to be on the safe side – do not pickle your entire training or testing functions for long-term storage. The pickle mechanism is aimed at for short-term storage, such as a temp file, or a copy to another machine in a distributed job.

Read more about serialization in Theano, or Python’s pickling.

3.5.2 Plotting Intermediate Results

Visualizations can be very powerful tools for understanding what your model or training algorithm is doing. You might be tempted to insert matplotlib plotting commands, or PIL image-rendering commands into your model-training script. However, later you will observe something interesting in one of those pre-rendered images and want to investigate something that isn’t clear from the pictures. You’ll wished you had saved the original model.

If you have enough disk space, your training script should save intermediate models and a visualization script should process those saved models.

You already have a model-saving function right? Just use it again to save these intermediate models.

Libraries you’ll want to know about: Python Image Library (PIL), matplotlib.
CHAPTER 4

Classifying MNIST digits using Logistic Regression

**Note:** This section assumes familiarity with the following Theano concepts: shared variables, basic arithmetic ops, T.grad, floatX. If you intend to run the code on GPU also read GPU.

**Note:** The code for this section is available for download here.

In this section, we show how Theano can be used to implement the most basic classifier: the logistic regression. We start off with a quick primer of the model, which serves both as a refresher but also to anchor the notation and show how mathematical expressions are mapped onto Theano graphs.

In the deepest of machine learning traditions, this tutorial will tackle the exciting problem of MNIST digit classification.

### 4.1 The Model

Logistic regression is a probabilistic, linear classifier. It is parametrized by a weight matrix $W$ and a bias vector $b$. Classification is done by projecting an input vector onto a set of hyperplanes, each of which corresponds to a class. The distance from the input to a hyperplane reflects the probability that the input is a member of the corresponding class.

Mathematically, the probability that an input vector $x$ is a member of a class $i$, a value of a stochastic variable $Y$, can be written as:

$$P(Y = i | x, W, b) = softmax_x(Wx + b)$$

$$= \frac{e^{Wix+b_i}}{\sum_j e^{W_jx+b_j}}$$
The model’s prediction \( y_{\text{pred}} \) is the class whose probability is maximal, specifically:

\[
y_{\text{pred}} = \arg\max_i P(Y = i | x, W, b)
\]

The code to do this in Theano is the following:

```python
# initialize with 0 the weights W as a matrix of shape (n_in, n_out)
self.W = theano.shared(
    value=numpy.zeros((n_in, n_out),
                      dtype=theano.config.floatX)
    ,
    name='W',
    borrow=True)

# initialize the biases b as a vector of n_out 0s
self.b = theano.shared(
    value=numpy.zeros((n_out,),
                      dtype=theano.config.floatX)
    ,
    name='b',
    borrow=True)

# symbolic expression for computing the matrix of class-membership
# probabilities
# Where:
# W is a matrix where column-k represent the separation hyperplane for
# class-k
# x is a matrix where row-j represents input training sample-j
# b is a vector where element-k represent the free parameter of
# hyperplane-k
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

# symbolic description of how to compute prediction as class whose
# probability is maximal
self.y_pred = T.argmax(self.p_y_given_x, axis=1)
```

Since the parameters of the model must maintain a persistent state throughout training, we allocate shared variables for \( W, b \). This declares them both as being symbolic Theano variables, but also initializes their contents. The dot and softmax operators are then used to compute the vector \( P(Y|x, W, b) \). The result \( p_{y \_given \_x} \) is a symbolic variable of vector-type.

To get the actual model prediction, we can use the \( \text{T.argmax} \) operator, which will return the index at which \( p_{y \_given \_x} \) is maximal (i.e. the class with maximum probability).

Now of course, the model we have defined so far does not do anything useful yet, since its parameters are still in their initial state. The following section will thus cover how to learn the optimal parameters.

**Note:** For a complete list of Theano ops, see: list of ops
4.2 Defining a Loss Function

Learning optimal model parameters involves minimizing a loss function. In the case of multi-class logistic regression, it is very common to use the negative log-likelihood as the loss. This is equivalent to maximizing the likelihood of the data set $\mathcal{D}$ under the model parameterized by $\theta$. Let us first start by defining the likelihood $\mathcal{L}$ and loss $\ell$:

$$
\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)}|x^{(i)}, W, b))
$$

$$
\ell(\theta = \{W, b\}, \mathcal{D}) = -\mathcal{L}(\theta = \{W, b\}, \mathcal{D})
$$

While entire books are dedicated to the topic of minimization, gradient descent is by far the simplest method for minimizing arbitrary non-linear functions. This tutorial will use the method of stochastic gradient method with mini-batches (MSGD). See Stochastic Gradient Descent for more details.

The following Theano code defines the (symbolic) loss for a given minibatch:

```python
# y.shape[0] is (symbolically) the number of rows in y, i.e.,
# number of examples (call it n) in the minibatch
# T.arange(y.shape[0]) is a symbolic vector which will contain
# [0,1,2,... n-1] T.log(self.p_y_given_x) is a matrix of
# Log-Probabilities (call it LP) with one row per example and
# one column per class LP[T.arange(y.shape[0]),y] is a vector
# v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,
# LP[n-1,y[n-1]]] and T.mean(LP[T.arange(y.shape[0]),y]) is
# the mean (across minibatch examples) of the elements in v,
# i.e., the mean log-likelihood across the minibatch.
return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
```

**Note:** Even though the loss is formally defined as the sum, over the data set, of individual error terms, in practice, we use the mean (T.mean) in the code. This allows for the learning rate choice to be less dependent of the minibatch size.

4.3 Creating a LogisticRegression class

We now have all the tools we need to define a LogisticRegression class, which encapsulates the basic behaviour of logistic regression. The code is very similar to what we have covered so far, and should be self explanatory.

```python
class LogisticRegression(object):  
    """Multi-class Logistic Regression Class
    
    The logistic regression is fully described by a weight matrix :math:`W`
    and bias vector :math:`b`. Classification is done by projecting data
    points onto a set of hyperplanes, the distance to which is used to
    determine a class membership probability.
    """
```

4.2. Defining a Loss Function 19
def __init__(self, input, n_in, n_out):
    """ Initialize the parameters of the logistic regression """
    :type input: theano.tensor.TensorType
    :param input: symbolic variable that describes the input of the architecture (one minibatch)
    
    :type n_in: int
    :param n_in: number of input units, the dimension of the space in which the datapoints lie
    
    :type n_out: int
    :param n_out: number of output units, the dimension of the space in which the labels lie
    
    """
    # start-snippet-1
    # initialize with 0 the weights W as a matrix of shape (n_in, n_out)
    self.W = theano.shared(
        value=numpy.zeros((n_in, n_out),
                          dtype=theano.config.floatX),
        name='W',
        borrow=True)
    # initialize the biases b as a vector of n_out 0s
    self.b = theano.shared(
        value=numpy.zeros((n_out,),
                          dtype=theano.config.floatX),
        name='b',
        borrow=True)
    # symbolic expression for computing the matrix of class-membership probabilities
    # Where:
    # W is a matrix where column-k represent the separation hyperplane for class-k
    # x is a matrix where row-j represents input training sample-j
    # b is a vector where element-k represent the free parameter of hyperplane-k
    self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
    # symbolic description of how to compute prediction as class whose probability is maximal
    self.y_pred = T.argmax(self.p_y_given_x, axis=1)
    # end-snippet-1

    # parameters of the model
self.params = [self.W, self.b]

# keep track of model input
self.input = input

def negative_log_likelihood(self, y):
    """Return the mean of the negative log-likelihood of the prediction of this model under a given target distribution.
    .. math::
        \frac{1}{|\mathcal{D}|} \mathcal{L} \left( \theta=\{W,b\}, \mathcal{D} \right)
    \rightarrow
        \frac{1}{|\mathcal{D}|} \sum_{i=0}^{|\mathcal{D}|} \log(P(Y=y^{(i)}|x^{(i)}, W,b)) \ell \left( \theta=\{W,b\}, \mathcal{D} \right)
    
    :type y: theano.tensor.TensorType
    :param y: corresponds to a vector that gives for each example the correct label

    Note: we use the mean instead of the sum so that the learning rate is less dependent on the batch size
    """
    # start-snippet-2
    # y.shape[0] is (symbolically) the number of rows in y, i.e.,
    # number of examples (call it n) in the minibatch
    # T.arange(y.shape[0]) is a symbolic vector which will contain
    # [0,1,2,... n-1] T.log(self.p_y_given_x) is a matrix of
    # Log-Probabilities (call it LP) with one row per example and
    # one column per class LP[T.arange(y.shape[0]),y] is a vector
    # v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,
    # LP[n-1,y[n-1]]] and T.mean(LP[T.arange(y.shape[0]),y]) is
    # the mean (across minibatch examples) of the elements in v,
    # i.e., the mean log-likelihood across the minibatch.
    return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
    # end-snippet-2

def errors(self, y):
    """Return a float representing the number of errors in the minibatch over the total number of examples of the minibatch ; zero one loss over the size of the minibatch
    
    :type y: theano.tensor.TensorType
    :param y: corresponds to a vector that gives for each example the correct label
    """
    # check if y has same dimension of y_pred
    if y.ndim != self.y_pred.ndim:
        raise TypeError('y should have the same shape as self.y_pred', ('y', y.type, 'y_pred', self.y_pred.type))

4.3. Creating a LogisticRegression class
We instantiate this class as follows:

```python
# generate symbolic variables for input (x and y represent a minibatch)
# data, presented as rasterized images
x = T.matrix('x')
# labels, presented as 1D vector of [int] labels
y = T.ivector('y')

# construct the logistic regression class
# Each MNIST image has size 28*28
classifier = LogisticRegression(input=x, n_in=28 * 28, n_out=10)
```

We start by allocating symbolic variables for the training inputs \( x \) and their corresponding classes \( y \). Note that \( x \) and \( y \) are defined outside the scope of the `LogisticRegression` object. Since the class requires the input to build its graph, it is passed as a parameter of the `__init__` function. This is useful in case you want to connect instances of such classes to form a deep network. The output of one layer can be passed as the input of the layer above. (This tutorial does not build a multi-layer network, but this code will be reused in future tutorials that do.)

Finally, we define a (symbolic) cost variable to minimize, using the instance method `classifier.negative_log_likelihood`.

```python
# the cost we minimize during training is the negative log likelihood of
# the model in symbolic format
cost = classifier.negative_log_likelihood(y)
```

Note that \( x \) is an implicit symbolic input to the definition of `cost`, because the symbolic variables of `classifier` were defined in terms of \( x \) at initialization.

### 4.4 Learning the Model

To implement MSGD in most programming languages (C/C++, Matlab, Python), one would start by manually deriving the expressions for the gradient of the loss with respect to the parameters: in this case \( \partial \ell / \partial W \) and \( \partial \ell / \partial b \). This can get pretty tricky for complex models, as expressions for \( \partial \ell / \partial \theta \) can get fairly complex, especially when taking into account problems of numerical stability.

With Theano, this work is greatly simplified. It performs automatic differentiation and applies certain math transforms to improve numerical stability.
To get the gradients $\partial \ell / \partial W$ and $\partial \ell / \partial b$ in Theano, simply do the following:

```python
    g_W = T.grad(cost=cost, wrt=classifier.W)
    g_b = T.grad(cost=cost, wrt=classifier.b)
```

$g_W$ and $g_b$ are symbolic variables, which can be used as part of a computation graph. The function `train_model`, which performs one step of gradient descent, can then be defined as follows:

```python
# specify how to update the parameters of the model as a list of
# (variable, update expression) pairs.
updates = [(classifier.W, classifier.W - learning_rate * g_W),
           (classifier.b, classifier.b - learning_rate * g_b)]

# compiling a Theano function `train_model` that returns the cost, but in
# the same time updates the parameter of the model based on the rules
# defined in `updates`
train_model = theano.function(input=[index],
                              outputs=cost,
                              updates=updates,
                              givens={
                                  x: train_set_x[index * batch_size: (index + 1) * batch_size],
                                  y: train_set_y[index * batch_size: (index + 1) * batch_size]
                              })
```

`updates` is a list of pairs. In each pair, the first element is the symbolic variable to be updated in the step, and the second element is the symbolic function for calculating its new value. Similarly, `givens` is a dictionary whose keys are symbolic variables and whose values specify their replacements during the step.

The function `train_model` is then defined such that:

- the input is the mini-batch index `index` that, together with the batch size (which is not an input since it is fixed) defines $x$ with corresponding labels $y$
- the return value is the cost/loss associated with the $x, y$ defined by the `index`
- on every function call, it will first replace $x$ and $y$ with the slices from the training set specified by `index`. Then, it will evaluate the cost associated with that minibatch and apply the operations defined by the `updates` list.

Each time `train_model(index)` is called, it will thus compute and return the cost of a minibatch, while also performing a step of MSGD. The entire learning algorithm thus consists in looping over all examples in the dataset, considering all the examples in one minibatch at a time, and repeatedly calling the `train_model` function.

### 4.5 Testing the model

As explained in *Learning a Classifier*, when testing the model we are interested in the number of misclassified examples (and not only in the likelihood). The `LogisticRegression` class therefore has an extra
instance method, which builds the symbolic graph for retrieving the number of misclassified examples in each minibatch.

The code is as follows:

```python
def errors(self, y):
    """Return a float representing the number of errors in the minibatch over the total number of examples of the minibatch; zero one loss over the size of the minibatch.

:type y: theano.tensor.TensorType
:param y: corresponds to a vector that gives for each example the correct label
    ""

    # check if y has same dimension of y_pred
    if y.ndim != self.y_pred.ndim:
        raise TypeError('y should have the same shape as self.y_pred', ('y', y.type, 'y_pred', self.y_pred.type))
    # check if y is of the correct datatype
    if y.dtype.startswith('int '):
        return T.mean(T.neq(self.y_pred, y))
    else:
        raise NotImplementedError()
```

We then create a function `test_model` and a function `validate_model`, which we can call to retrieve this value. As you will see shortly, `validate_model` is key to our early-stopping implementation (see Early-Stopping). These functions take a minibatch index and compute, for the examples in that minibatch, the number that were misclassified by the model. The only difference between them is that `test_model` draws its minibatches from the testing set, while `validate_model` draws its from the validation set.

```python
# compiling a Theano function that computes the mistakes that are made by
# the model on a minibatch

test_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)
```
4.6 Putting it All Together

The finished product is as follows.

"""
This tutorial introduces logistic regression using Theano and stochastic gradient descent.

Logistic regression is a probabilistic, linear classifier. It is parametrized by a weight matrix :math:`W` and a bias vector :math:`b`. Classification is done by projecting data points onto a set of hyperplanes, the distance to which is used to determine a class membership probability.

Mathematically, this can be written as:

.. math::
   P(Y=i|x, W, b) &= \text{softmax}_i(W x + b) \\
   &= \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}

The output of the model or prediction is then done by taking the argmax of the vector whose i'th element is :math:`P(Y=i|x)`.

.. math::
   y_{pred} = \text{argmax}_i P(Y=i|x, W, b)

This tutorial presents a stochastic gradient descent optimization method suitable for large datasets.

References:

- textbooks: "Pattern Recognition and Machine Learning" - Christopher M. Bishop, section 4.3.2

"""

from __future__ import print_function
__docformat__ = 'restructuredtext en'

import six.moves.cPickle as pickle
import gzip
import os
import sys
import timeit
import numpy
import theano
import theano.tensor as T

class LogisticRegression(object):
    '''Multi-class Logistic Regression Class

    The logistic regression is fully described by a weight matrix $W$ and
    bias vector $b$. Classification is done by projecting data points onto
    a set of hyperplanes, the distance to which is used to determine a
    class membership probability.
    '''

    def __init__(self, input, n_in, n_out):
        '''Initialize the parameters of the logistic regression

        :type input: theano.tensor.TensorType
        :param input: symbolic variable that describes the input of the
                      architecture (one minibatch)

        :type n_in: int
        :param n_in: number of input units, the dimension of the space in
                     which the datapoints lie

        :type n_out: int
        :param n_out: number of output units, the dimension of the space in
                      which the labels lie
        '''

        # start-snippet-1
        # initialize with 0 the weights W as a matrix of shape (n_in, n_out)
        self.W = theano.shared(
            value=numpy.zeros(
                (n_in, n_out),
                dtype=theano.config.floatX
            ),
            name='W',
            borrow=True
        )

        # initialize the biases b as a vector of n_out 0s
        self.b = theano.shared(
            value=numpy.zeros(
                (n_out,),
                dtype=theano.config.floatX
            ),
            name='b',
            borrow=True
        )

Chapter 4. Classifying MNIST digits using Logistic Regression
# symbolic expression for computing the matrix of class-membership
# probabilities
# Where:
# W is a matrix where column-k represent the separation hyperplane for
class-k
# x is a matrix where row-j represents input training sample-j
# b is a vector where element-k represent the free parameter of
# hyperplane-k
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

# symbolic description of how to compute prediction as class whose
# probability is maximal
self.y_pred = T.argmax(self.p_y_given_x, axis=1)
# end-snippet-1

# parameters of the model
self.params = [self.W, self.b]

# keep track of model input
self.input = input

def negative_log_likelihood(self, y):
    """Return the mean of the negative log-likelihood of the prediction
of this model under a given target distribution.
    .. math::
        \frac{1}{|\mathcal{D}|} \mathcal{L} (\theta=\{W,b\}, \mathcal{D})
        \rightarrow
        \frac{1}{|\mathcal{D}|} \sum_{i=0}^{|\mathcal{D}|} \log(P(Y=y^{(i)}|x^{(i)}, W,b)) \ell (\theta=\{W,b\}, \mathcal{D})
    :type y: theano.tensor.TensorType
    :param y: corresponds to a vector that gives for each example the
correct label

    Note: we use the mean instead of the sum so that
    the learning rate is less dependent on the batch size
    ""
    # start-snippet-2
    # y.shape[0] is (symbolically) the number of rows in y, i.e.,
    # number of examples (call it n) in the minibatch
    # T.arange(y.shape[0]) is a symbolic vector which will contain
    # [0,1,2,... n-1] T.log(self.p_y_given_x) is a matrix of
    # Log-Probabilities (call it LP) with one row per example and
    # one column per class LP[T.arange(y.shape[0]),y] is a vector
    # v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,
    # LP[n-1,y[n-1]]] and T.mean(LP[T.arange(y.shape[0]),y]) is
    # the mean (across minibatch examples) of the elements in v,
    # i.e., the mean log-likelihood across the minibatch.
    return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
    # end-snippet-2
def errors(self, y):
    """Return a float representing the number of errors in the minibatch
    over the total number of examples of the minibatch; zero one
    loss over the size of the minibatch
    :type y: theano.tensor.TensorType
    :param y: corresponds to a vector that gives for each example the
              correct label
    ""
    # check if y has same dimension of y_pred
    if y.ndim != self.y_pred.ndim:
        raise TypeError('y should have the same shape as self.y_pred',
                        ('y', y.type, 'y_pred', self.y_pred.type))
    # check if y is of the correct datatype
    if y.dtype.startswith('int'):
        # the T.neq operator returns a vector of 0s and 1s, where 1
        # represents a mistake in prediction
        return T.mean(T.neq(self.y_pred, y))
    else:
        raise NotImplementedError()

def load_data(dataset):
    """Loads the dataset
    :type dataset: string
    :param dataset: the path to the dataset (here MNIST)
    ""
    #############
    # LOAD DATA #
    #############

    # Download the MNIST dataset if it is not present
    data_dir, data_file = os.path.split(dataset)
    if data_dir == '' and not os.path.isfile(dataset):
        # Check if dataset is in the data directory.
        new_path = os.path.join(
            os.path.split(__file__)[0],
            '..',
            'data',
            dataset
        )
        if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
            dataset = new_path

    if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
        from six.moves import urllib
        origin = (
Downloading data from %s
urllib.request.urlretrieve(origin, dataset)

print('... loading data')

# Load the dataset
with gzip.open(dataset, 'rb') as f:
    try:
        train_set, valid_set, test_set = pickle.load(f, encoding='latin1')
    except:
        train_set, valid_set, test_set = pickle.load(f)

# train_set, valid_set, test_set format: tuple(input, target)
# input is a numpy.ndarray of 2 dimensions (a matrix)
# where each row corresponds to an example. target is a
# numpy.ndarray of 1 dimension (vector) that has the same length as
# the number of rows in the input. It should give the target
# to the example with the same index in the input.

def shared_dataset(data_xy, borrow=True):
    """ Function that loads the dataset into shared variables

    The reason we store our dataset in shared variables is to allow
    Theano to copy it into the GPU memory (when code is run on GPU).
    Since copying data into the GPU is slow, copying a minibatch everytime
    is needed (the default behaviour if the data is not in a shared
    variable) would lead to a large decrease in performance.
    ""

data_x, data_y = data_xy
shared_x = theano.shared(numpy.asarray(data_x,
                                        dtype=theano.config.floatX),
                         borrow=borrow)
shared_y = theano.shared(numpy.asarray(data_y,
                                        dtype=theano.config.floatX),
                         borrow=borrow)

# When storing data on the GPU it has to be stored as floats
# therefore we will store the labels as ``floatX'' as well
# (``shared_y`` does exactly that). But during our computations
# we need them as ints (we use labels as index, and if they are
# floats it doesn't make sense) therefore instead of returning
# ``shared_y`` we will have to cast it to int. This little hack
# lets us get around this issue
return shared_x, T.cast(shared_y, 'int32')

test_set_x, test_set_y = shared_dataset(test_set)
valid_set_x, valid_set_y = shared_dataset(valid_set)
train_set_x, train_set_y = shared_dataset(train_set)

rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
        (test_set_x, test_set_y)]
return rval

4.6. Putting it All Together
```python
def sgd_optimization_mnist(learning_rate=0.13, n_epochs=1000, 
dataset='mnist.pkl.gz', 
batch_size=600):
    
    """
    Demonstrate stochastic gradient descent optimization of a log-linear model
    
    This is demonstrated on MNIST.
    :type learning_rate: float
    :param learning_rate: learning rate used (factor for the stochastic gradient)
    
    :type n_epochs: int
    :param n_epochs: maximal number of epochs to run the optimizer
    
    :type dataset: string
    :param dataset: the path of the MNIST dataset file from
    http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz
    """

    datasets = load_data(dataset)

    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    # compute number of minibatches for training, validation and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] // batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size

    ######################
    # BUILD ACTUAL MODEL #
    ######################
    print('... building the model')

    # allocate symbolic variables for the data
    index = T.lscalar()  # index to a [mini]batch

    # generate symbolic variables for input (x and y represent a minibatch)
    x = T.matrix('x')  # data, presented as rasterized images
    y = T.ivector('y')  # labels, presented as 1D vector of [int] labels

    # construct the logistic regression class
    # Each MNIST image has size 28*28
    classifier = LogisticRegression(input=x, n_in=28 * 28, n_out=10)
```

Chapter 4. Classifying MNIST digits using Logistic Regression
# the cost we minimize during training is the negative log likelihood of
# the model in symbolic format

cost = classifier.negative_log_likelihood(y)

# compiling a Theano function that computes the mistakes that are made by
# the model on a minibatch
test_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# compute the gradient of cost with respect to theta = (W,b)
g_W = T.grad(cost=cost, wrt=classifier.W)
g_b = T.grad(cost=cost, wrt=classifier.b)

# start-snippet-3
# specify how to update the parameters of the model as a list of
# (variable, update expression) pairs.
updates = [(classifier.W, classifier.W - learning_rate * g_W),
           (classifier.b, classifier.b - learning_rate * g_b)]

# compiling a Theano function 'train_model' that returns the cost, but in
# the same time updates the parameter of the model based on the rules
# defined in `updates`
train_model = theano.function(
    inputs=[index],
    outputs=cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]
    }
)
# end-snippet-3

###############
# TRAIN MODEL#
###############

print('... training the model')
patience = 5000  # look as this many examples regardless
patience_increase = 2  # wait this much longer when a new best is
# found
improvement_threshold = 0.995  # a relative improvement of this much is
# considered significant
validation_frequency = min(n_train_batches, patience // 2)
# go through this many
# minibatche before checking the network
# on the validation set; in this case we
# check every epoch

best_validation_loss = numpy.inf
test_score = 0.
start_time = timeit.default_timer()

done_looping = False
epoch = 0
while (epoch < n_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in range(n_train_batches):
        minibatch_avg_cost = train_model(minibatch_index)
        iter = (epoch - 1) * n_train_batches + minibatch_index

        if (iter + 1) % validation_frequency == 0:
            # compute zero-one loss on validation set
            validation_losses = [validate_model(i) for i in range(n_valid_batches)]
            this_validation_loss = numpy.mean(validation_losses)
            print('epoch %i, minibatch %i/%i, validation error %f %%' %
                  (epoch, minibatch_index + 1, n_train_batches,
                   this_validation_loss * 100.))

            # if we got the best validation score until now
            if this_validation_loss < best_validation_loss:
                # improve patience if loss improvement is good enough
                if this_validation_loss < best_validation_loss * improvement_threshold:
                    patience = max(patience, iter * patience_increase)

                best_validation_loss = this_validation_loss
                # test it on the test set
                test_losses = [test_model(i) for i in range(n_test_batches)]
test_score = numpy.mean(test_losses)

print(
    '{: epoch $i$, minibatch $i/\$i$, test error of' +
    ' best model $f$ %%'
) %

    epoch,
    minibatch_index + 1,
    n_train_batches,
    test_score * 100.
)

# save the best model

with open('best_model.pkl', 'wb') as f:
    pickle.dump(classifier, f)

if patience <= iter:
    done_looping = True
    break

end_time = timeit.default_timer()

print{
    'Optimization complete with best validation score of $f$ %,'
    'with test performance $f$ %'
) % (best_validation_loss * 100., test_score * 100.)

print('The code run for %d epochs, with %f epochs/sec' % (epoch, 1. * epoch / (end_time - start_time)))

print(('The code for file ' +
    os.path.split(__file__) [1] +
    ' ran for %.1fs' % ((end_time - start_time)), file=sys.stderr)

def predict():
    """
    An example of how to load a trained model and use it
    to predict labels.
    """

    # load the saved model
    classifier = pickle.load(open('best_model.pkl'))

    # compile a predictor function
    predict_model = theano.function(
        inputs=[classifier.input],
        outputs=classifier.y_pred)

    # We can test it on some examples from test test
dataset='mnist.pkl.gz'
datasets = load_data(dataset)
test_set_x, test_set_y = datasets[2]
test_set_x = test_set_x.get_value()

predicted_values = predict_model(test_set_x[:10])
print("Predicted values for the first 10 examples in test set:")
print(predicted_values)

if __name__ == '__main__':
    sgd_optimization_mnist()  

The user can learn to classify MNIST digits with SGD logistic regression, by typing, from within the DeepLearningTutorials folder:

python code/logistic_sgd.py

The output one should expect is of the form:

...  
epoch 72, minibatch 83/83, validation error 7.510417 %
  epoch 72, minibatch 83/83, test error of best model 7.510417 %
epoch 73, minibatch 83/83, validation error 7.500000 %
  epoch 73, minibatch 83/83, test error of best model 7.489583 %
Optimization complete with best validation score of 7.500000 %, with test
  performance 7.489583 %
The code run for 74 epochs, with 1.936983 epochs/sec

On an Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00 Ghz the code runs with approximately 1.936 epochs/sec and it took 75 epochs to reach a test error of 7.489%. On the GPU the code does almost 10.0 epochs/sec. For this instance we used a batch size of 600.

4.7 Prediction Using a Trained Model

sgd_optimization_mnist serialize and pickle the model each time new lowest validation error is reached. We can reload this model and predict labels of new data. predict function shows an example of how this could be done.

def predict():
    ""
    An example of how to load a trained model and use it to predict labels.
    ""
    
    # load the saved model
    classifier = pickle.load(open('best_model.pkl'))

    # compile a predictor function
    predict_model = theano.function(  

    ...
# We can test it on some examples from test test
dataset='mnist.pkl.gz'
datasets = load_data(dataset)
test_set_x, test_set_y = datasets[2]
test_set_x = test_set_x.get_value()

predicted_values = predict_model(test_set_x[:10])
print("Predicted values for the first 10 examples in test set:")
print(predicted_values)
CHAPTER 5

Multilayer Perceptron

Note: This section assumes the reader has already read through *Classifying MNIST digits using Logistic Regression*. Additionally, it uses the following new Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, *L1 and L2 regularization*, floatX. If you intend to run the code on GPU also read GPU.

Note: The code for this section is available for download here.

The next architecture we are going to present using Theano is the single-hidden-layer Multi-Layer Perceptron (MLP). An MLP can be viewed as a logistic regression classifier where the input is first transformed using a learnt non-linear transformation \( \Phi \). This transformation projects the input data into a space where it becomes linearly separable. This intermediate layer is referred to as a **hidden layer**. A single hidden layer is sufficient to make MLPs a **universal approximator**. However we will see later on that there are substantial benefits to using many such hidden layers, i.e. the very premise of **deep learning**. See these course notes for an introduction to MLPs, the back-propagation algorithm, and how to train MLPs.

This tutorial will again tackle the problem of MNIST digit classification.

5.1 The Model

An MLP (or Artificial Neural Network - ANN) with a single hidden layer can be represented graphically as follows:

Formally, a one-hidden-layer MLP is a function \( f : \mathbb{R}^D \rightarrow \mathbb{R}^L \), where \( D \) is the size of input vector \( x \) and \( L \)
is the size of the output vector \( f(x) \), such that, in matrix notation:

\[
f(x) = G(b^{(2)} + W^{(2)} (s(b^{(1)} + W^{(1)}x))),
\]

with bias vectors \( b^{(1)}, b^{(2)} \); weight matrices \( W^{(1)}, W^{(2)} \) and activation functions \( G \) and \( s \).

The vector \( h(x) = \Phi(x) = s(b^{(1)} + W^{(1)}x) \) constitutes the hidden layer. \( W^{(1)} \in R^{D \times D_h} \) is the weight matrix connecting the input vector to the hidden layer. Each column \( W^{(1)}_i \) represents the weights from the input units to the \( i \)-th hidden unit. Typical choices for \( s \) include \( \tanh \), with \( \tanh(a) = (e^a - e^{-a})/(e^a + e^{-a}) \), or the logistic \( \text{sigmoid} \) function, with \( \text{sigmoid}(a) = 1/(1 + e^{-a}) \). We will be using \( \tanh \) in this tutorial because it typically yields to faster training (and sometimes also to better local minima). Both the \( \tanh \) and \( \text{sigmoid} \) are scalar-to-scalar functions but their natural extension to vectors and tensors consists in applying them element-wise (e.g. separately on each element of the vector, yielding a same-size vector).

The output vector is then obtained as: \( o(x) = G(b^{(2)} + W^{(2)}h(x)) \). The reader should recognize the form we already used for Classifying MNIST digits using Logistic Regression. As before, class-membership probabilities can be obtained by choosing \( G \) as the \textit{softmax} function (in the case of multi-class classification).

To train an MLP, we learn all parameters of the model, and here we use \textit{Stochastic Gradient Descent} with minibatches. The set of parameters to learn is the set \( \theta = \{W^{(2)}, b^{(2)}, W^{(1)}, b^{(1)}\} \). Obtaining the gradients \( \partial \ell / \partial \theta \) can be achieved through the \textit{backpropagation algorithm} (a special case of the chain-rule of derivation). Thankfully, since Theano performs automatic differentiation, we will not need to cover this in the tutorial!

### 5.2 Going from logistic regression to MLP

This tutorial will focus on a single-hidden-layer MLP. We start off by implementing a class that will represent a hidden layer. To construct the MLP we will then only need to throw a logistic regression layer on top.

```python
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):
        """
        Typical hidden layer of a MLP: units are fully-connected and have sigmoidal activation function. Weight matrix W is of shape (n_in,n_out) and the bias vector b is of shape (n_out,).
        """
```
NOTE: The nonlinearity used here is tanh

Hidden unit activation is given by: \( \text{tanh}(\text{dot}(\text{input}, W) + b) \)

:type rng: numpy.random.RandomState
:param rng: a random number generator used to initialize weights

type input: theano.tensor.dmatrix
:param input: a symbolic tensor of shape (n_examples, n_in)

type n_in: int
:param n_in: dimensionality of input

type n_out: int
:param n_out: number of hidden units

type activation: theano.Op or function
:param activation: Non linearity to be applied in the hidden layer

```
self.input = input
```

The initial values for the weights of a hidden layer \( i \) should be uniformly sampled from a symmetric interval that depends on the activation function. For \( \text{tanh} \) activation function results obtained in [Xavier10] show that the interval should be \( \left[ -\sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}}, \sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}} \right] \), where \( \text{fan}_{in} \) is the number of units in the \( (i-1) \)-th layer, and \( \text{fan}_{out} \) is the number of units in the \( i \)-th layer. For the sigmoid function the interval is \( \left[ -4\sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}}, 4\sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}} \right] \). This initialization ensures that, early in training, each neuron operates in a regime of its activation function where information can easily be propagated both upward (activations flowing from inputs to outputs) and backward (gradients flowing from outputs to inputs).

```python
# W is initialized with W_values which is uniformly sampled
# from sqrt(-6./((n_in+n_hidden))) and sqrt(6./((n_in+n_hidden)))
# for tanh activation function
# the output of uniform if converted using asarray to dtype
# theano.config.floatX so that the code is runable on GPU
# Note: optimal initialization of weights is dependent on the
# activation function used (among other things).
# For example, results presented in [Xavier10] suggest that you
# should use 4 times larger initial weights for sigmoid
# compared to tanh.
# We have no info for other function, so we use the same as
# tanh.
if W is None:
    W_values = numpy.asarray(
        rng.uniform(
            low=-numpy.sqrt(6. / (n_in + n_out)),
            high=numpy.sqrt(6. / (n_in + n_out)),
            size=(n_in, n_out)
        ),
        dtype=theano.config.floatX
```

5.2. Going from logistic regression to MLP 39
if activation == theano.tensor.nnet.sigmoid:
    W_values *= 4

W = theano.shared(value=W_values, name='W', borrow=True)

if b is None:
    b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
    b = theano.shared(value=b_values, name='b', borrow=True)

self.W = W
self.b = b

Note that we used a given non-linear function as the activation function of the hidden layer. By default this
is \text{tanh}, but in many cases we might want to use something else.

lin_output = T.dot(input, self.W) + self.b
self.output = (lin_output if activation is None else activation(lin_output))

If you look into theory this class implements the graph that computes the hidden layer value \( h(x) = \Phi(x) = s(b_1 + W_1 x) \). If you give this graph as input to the \texttt{LogisticRegression} class, implemented in the previous tutorial \textit{Classifying MNIST digits using Logistic Regression}, you get the output of the MLP. You can see this in the following short implementation of the \texttt{MLP} class.

class MLP(object):
    """Multi-Layer Perceptron Class

    A multilayer perceptron is a feedforward artificial neural network model that has one layer or more of hidden units and nonlinear activations. Intermediate layers usually have as activation function tanh or the sigmoid function (defined here by a `HiddenLayer` class) while the top layer is a softmax layer (defined here by a `LogisticRegression` class).
    """

    def __init__(self, rng, input, n_in, n_hidden, n_out):
        """Initialize the parameters for the multilayer perceptron

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.TensorType
        :param input: symbolic variable that describes the input of the architecture (one minibatch)

        :type n_in: int
        :param n_in: number of input units, the dimension of the space in
which the datapoints lie

:type n_hidden: int
:param n_hidden: number of hidden units

:type n_out: int
:param n_out: number of output units, the dimension of the space in
which the labels lie

""

# Since we are dealing with a one hidden layer MLP, this will
→ translate
# into a HiddenLayer with a tanh activation function connected to the
# LogisticRegression layer; the activation function can be replaced by
# sigmoid or any other nonlinear function
self.hiddenLayer = HiddenLayer(
    rng=rng,
    input=input,
    n_in=n_in,
    n_out=n_hidden,
    activation=T.tanh
)

# The logistic regression layer gets as input the hidden units
# of the hidden layer
self.logRegressionLayer = LogisticRegression(
    input=self.hiddenLayer.output,
    n_in=n_hidden,
    n_out=n_out
)

In this tutorial we will also use L1 and L2 regularization (see L1 and L2 regularization). For this, we need
to compute the L1 norm and the squared L2 norm of the weights $W^{(1)}, W^{(2)}$.

# L1 norm ; one regularization option is to enforce L1 norm to
# be small
self.L1 = (
    abs(self.hiddenLayer.W).sum()
    + abs(self.logRegressionLayer.W).sum()
)

# square of L2 norm ; one regularization option is to enforce
# square of L2 norm to be small
self.L2_sqr = (n
    (self.hiddenLayer.W ** 2).sum()
    + (self.logRegressionLayer.W ** 2).sum()
)

# negative log likelihood of the MLP is given by the negative
# log likelihood of the output of the model, computed in the
# logistic regression layer
self.negative_log_likelihood = (}
As before, we train this model using stochastic gradient descent with mini-batches. The difference is that we modify the cost function to include the regularization term. L1_reg and L2_reg are the hyperparameters controlling the weight of these regularization terms in the total cost function. The code that computes the new cost is:

```python
# the cost we minimize during training is the negative log likelihood of # the model plus the regularization terms (L1 and L2); cost is expressed # here symbolically
cost = (
    classifier.negative_log_likelihood(y) + L1_reg * classifier.L1 + L2_reg * classifier.L2_sqr
)
```

We then update the parameters of the model using the gradient. This code is almost identical to the one for logistic regression. Only the number of parameters differ. To get around this (and write code that could work for any number of parameters) we will use the list of parameters that we created with the model params and parse it, computing a gradient at each step.

```python
# compute the gradient of cost with respect to theta (sorted in params) # the resulting gradients will be stored in a list gparams
gparams = [T.grad(cost, param) for param in classifier.params]

# specify how to update the parameters of the model as a list of # (variable, update expression) pairs

given two lists of the same length, A = [a1, a2, a3, a4] and # B = [b1, b2, b3, b4], zip generates a list C of same size, where each # element is a pair formed from the two lists :
# C = [(a1, b1), (a2, b2), (a3, b3), (a4, b4)]
updates = [
    (param, param - learning_rate * gparam)
    for param, gparam in zip(classifier.params, gparams)
]

# compiling a Theano function `train_model` that returns the cost, but # in the same time updates the parameter of the model based on the rules # defined in `updates`
train_model = theano.function(
    inputs=[index],
    outputs=cost,
    updates=updates,
)
5.3 Putting it All Together

Having covered the basic concepts, writing an MLP class becomes quite easy. The code below shows how this can be done, in a way which is analogous to our previous logistic regression implementation.

```
""
This tutorial introduces the multilayer perceptron using Theano.

A multilayer perceptron is a logistic regressor where instead of feeding the input to the logistic regression you insert a intermediate layer, called the hidden layer, that has a nonlinear activation function (usually tanh or sigmoid). One can use many such hidden layers making the architecture deep. The tutorial will also tackle the problem of MNIST digit classification.

.. math::

\[ f(x) = G( b^{(2)} + W^{(2)}( s( b^{(1)} + W^{(1)} x))) , \]

References:
- textbooks: "Pattern Recognition and Machine Learning" - Christopher M. Bishop, section 5

""
```
from __future__ import print_function
__docformat__ = 'restructuredtext en'

import os
import sys
import timeit
import numpy
import theano
import theano.tensor as T

from logistic_sgd import LogisticRegression, load_data

# start-snippet-1

givens={
    x: train_set_x[index * batch_size: (index + 1) * batch_size],
    y: train_set_y[index * batch_size: (index + 1) * batch_size]
}
"""
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,  
                 activation=T.tanh):
        
        """
        Typical hidden layer of a MLP: units are fully-connected and have
        sigmoidal activation function. Weight matrix W is of shape (n_in,n_  
        →out)
        and the bias vector b is of shape (n_out,).
        
        NOTE : The nonlinearity used here is tanh
        
        Hidden unit activation is given by: tanh(dot(input,W) + b)
        """
        self.input = input

        # end-snippet-1

        # `W` is initialized with `W_values` which is uniformely sampled
        # from sqrt(-6./(n_in+n_hidden)) and sqrt(6./(n_in+n_hidden))
        # for tanh activation function
        # the output of uniform if converted using asarray to dtype
        # theano.config.floatX so that the code is runnable on GPU
        # Note : optimal initialization of weights is dependent on the
        # activation function used (among other things).
        # For example, results presented in [Xavier10] suggest that you
        # should use 4 times larger initial weights for sigmoid
        # compared to tanh
        # We have no info for other function, so we use the same as
        # tanh.
        if W is None:
            W_values = numpy.asarray(
                rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)
                ),
                dtype=theano.config.floatX
            )
\[
\text{if } \text{activation} == \text{theano.tensor.nnet.sigmoid}:
\]
\[
W_{\text{values}} *= 4
\]
\[
W = \text{theano.shared(value}=W_{\text{values}}, \text{name}='W', \text{borrow}=\text{True})
\]
\[
\text{if } \text{b is None}:
\]
\[
b_{\text{values}} = \text{numpy.zeros}((n_{\text{out}},)) \text{, dtype=theano.config.floatX})
\]
\[
b = \text{theano.shared(value}=b_{\text{values}}, \text{name}='b', \text{borrow}=\text{True})
\]
\[
\text{self}.W = W
\]
\[
\text{self}.b = b
\]
\[
\text{lin}_\text{output} = \text{T.dot(input, self.W)} + \text{self.b}
\]
\[
\text{self}.output = (\text{lin}_\text{output} \text{if } \text{activation} \text{is None} \text{else} \text{activation(lin}_\text{output)}
\]
\[
\text{# parameters of the model}
\]
\[
\text{self}.\text{params} = [\text{self}.W, \text{self}.b]
\]

\# start-snippet-2

class MLP(object):

    """Multi-Layer Perceptron Class
    A multilayer perceptron is a feedforward artificial neural network model
    that has one layer or more of hidden units and nonlinear activations.
    Intermediate layers usually have as activation function tanh or the
    sigmoid function (defined here by a `HiddenLayer`` class) while the
    top layer is a softmax layer (defined here by a `LogisticRegression``
    class).
    """

    def __init__(self, rng, input, n_in, n_hidden, n_out):
        """Initialize the parameters for the multilayer perceptron

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.TensorType
        :param input: a random number generator used to initialize weights

        :type n_in: int
        :param n_in: number of input units, the dimension of the space in
        which the datapoints lie

        :type n_hidden: int
        :param n_hidden: number of hidden units

        :type n_out: int
        :param n_out: number of output units, the dimension of the space in
        which the labels lie
        """
# Since we are dealing with a one hidden layer MLP, this will translate into a HiddenLayer with a tanh activation function connected to the LogisticRegression layer; the activation function can be replaced by sigmoid or any other nonlinear function
self.hiddenLayer = HiddenLayer(
    rng=rng,
    input=input,
    n_in=n_in,
    n_out=n_hidden,
    activation=T.tanh
)

# The logistic regression layer gets as input the hidden units of the hidden layer
self.logRegressionLayer = LogisticRegression(
    input=self.hiddenLayer.output,
    n_in=n_hidden,
    n_out=n_out
)

# L1 norm ; one regularization option is to enforce L1 norm to be small
self.L1 = (  
    abs(self.hiddenLayer.W).sum()  
    + abs(self.logRegressionLayer.W).sum()
)

# square of L2 norm ; one regularization option is to enforce square of L2 norm to be small
self.L2_sqr = (  
    (self.hiddenLayer.W ** 2).sum()  
    + (self.logRegressionLayer.W ** 2).sum()
)

# negative log likelihood of the MLP is given by the negative log likelihood of the output of the model, computed in the logistic regression layer
self.negative_log_likelihood = (  
    self.logRegressionLayer.negative_log_likelihood
)

# same holds for the function computing the number of errors
self.errors = self.logRegressionLayer.errors

# the parameters of the model are the parameters of the two layer it is made out of
self.params = self.hiddenLayer.params + self.logRegressionLayer.params

# end-snippet-3
# keep track of model input
self.input = input

def test_mlp(learning_rate=0.01, L1_reg=0.00, L2_reg=0.0001, n_epochs=1000,
             dataset='mnist.pkl.gz', batch_size=20, n_hidden=500):
    
    Demonstrate stochastic gradient descent optimization for a multilayer
    perceptron

    This is demonstrated on MNIST.

    :type learning_rate: float
    :param learning_rate: learning rate used (factor for the stochastic
    gradient

    :type L1_reg: float
    :param L1_reg: L1-norm's weight when added to the cost (see
    regularization)

    :type L2_reg: float
    :param L2_reg: L2-norm's weight when added to the cost (see
    regularization)

    :type n_epochs: int
    :param n_epochs: maximal number of epochs to run the optimizer

    :type dataset: string
    :param dataset: the path of the MNIST dataset file from

        http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz

    
    datasets = load_data(dataset)

    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    # compute number of minibatches for training, validation and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] // batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size

    ###########################
    # BUILD ACTUAL MODEL #
    ###########################
    print('... building the model')

    # allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch
x = T.matrix('x')  # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
# [int] labels

rng = numpy.random.RandomState(1234)

# construct the MLP class
classifier = MLP(
    rng=rng,
    input=x,
    n_in=28 * 28,
    n_hidden=n_hidden,
    n_out=10
)

# start-snippet-4
# the cost we minimize during training is the negative log likelihood of
# the model plus the regularization terms (L1 and L2); cost is expressed
# here symbolically
cost = (classifier.negative_log_likelihood(y)
        + L1_reg * classifier.L1
        + L2_reg * classifier.L2_sqr)
# end-snippet-4

# compiling a Theano function that computes the mistakes that are made
# by the model on a minibatch

validate_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: valid_set_x[index * batch_size:(index + 1) * batch_size],
        y: valid_set_y[index * batch_size:(index + 1) * batch_size]
    }
)

# start-snippet-5
# compute the gradient of cost with respect to theta (sorted in params)
# the resulting gradients will be stored in a list gparams

gparams = [T.grad(cost, param) for param in classifier.params]
# specify how to update the parameters of the model as a list of
# (variable, update expression) pairs

# given two lists of the same length, A = [a1, a2, a3, a4] and
# B = [b1, b2, b3, b4], zip generates a list C of same size, where each
# element is a pair formed from the two lists :
# C = [(a1, b1), (a2, b2), (a3, b3), (a4, b4)]
updates = [
    (param, param - learning_rate * gparam)
    for param, gparam in zip(classifier.params, gparams)
]

# compiling a Theano function `train_model` that returns the cost, but
# in the same time updates the parameter of the model based on the rules
# defined in `updates`
train_model = theano.function(
    inputs=[index],
    outputs=cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

###########
# TRAIN MODEL #
###########

print('... training')

# early-stopping parameters
patience = 10000 # look as this many examples regardless
patience_increase = 2 # wait this much longer when a new best is
# found
improvement_threshold = 0.995 # a relative improvement of this much is
# considered significant
validation_frequency = min(n_train_batches, patience // 2)
# go through this many
# minibatche before checking the network
# on the validation set; in this case we
# check every epoch

best_validation_loss = numpy.inf
best_iter = 0
start_time = timeit.default_timer()

epoch = 0
done_looping = False

while (epoch < n_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in range(n_train_batches):
        best_validation_loss = numpy.inf
        best_iter = 0
        start_time = timeit.default_timer()
minibatch_avg_cost = train_model(minibatch_index)
# iteration number
iter = (epoch - 1) * n_train_batches + minibatch_index

if (iter + 1) % validation_frequency == 0:
    # compute zero-one loss on validation set
    validation_losses = [validate_model(i) for i in range(n_valid_batches)]
    this_validation_loss = numpy.mean(validation_losses)

    print('epoch %i, minibatch %i/%i, validation error %f %%'
          (epoch, minibatch_index + 1, n_train_batches, this_validation_loss * 100.))

    # if we got the best validation score until now
    if this_validation_loss < best_validation_loss:
        # improve patience if loss improvement is good enough
        if (this_validation_loss < best_validation_loss * improvement_threshold):
            patience = max(patience, iter * patience_increase)

        best_validation_loss = this_validation_loss
        best_iter = iter

    # test it on the test set
    test_losses = [test_model(i) for i in range(n_test_batches)]
    test_score = numpy.mean(test_losses)

    print(('epoch %i, minibatch %i/%i, test error of ' 'best model %f %')
          (epoch, minibatch_index + 1, n_train_batches, test_score * 100.))

if patience <= iter:
    done_looping = True
    break

end_time = timeit.default_timer()
print(('Optimization complete. Best validation score of %f %%
       obtained at iteration %i, with test performance %f %%')
       (best_validation_loss * 100., best_iter + 1, test_score * 100.))
print(('The code for file ' + os.path.split(__file__)[1] +...  }
The user can then run the code by calling:

```
python code/mlp.py
```

The output one should expect is of the form:

```
Optimization complete. Best validation score of 1.690000 % obtained at
iteration 2070000, with test performance 1.650000 %
The code for file mlp.py ran for 97.34m
```

On an Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz the code runs with approximately 10.3 epoch/minute and it took 828 epochs to reach a test error of 1.65%.

To put this into perspective, we refer the reader to the results section of this page.

## 5.4 Tips and Tricks for training MLPs

There are several hyper-parameters in the above code, which are not (and, generally speaking, cannot be) optimized by gradient descent. Strictly speaking, finding an optimal set of values for these hyper-parameters is not a feasible problem. First, we can’t simply optimize each of them independently. Second, we cannot readily apply gradient techniques that we described previously (partly because some parameters are discrete values and others are real-valued). Third, the optimization problem is not convex and finding a (local) minimum would involve a non-trivial amount of work.

The good news is that over the last 25 years, researchers have devised various rules of thumb for choosing hyper-parameters in a neural network. A very good overview of these tricks can be found in *Efficient BackProp* by Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus-Robert Mueller. In here, we summarize the same issues, with an emphasis on the parameters and techniques that we actually used in our code.

### 5.4.1 Nonlinearity

Two of the most common ones are the *sigmoid* and the *tanh* function. For reasons explained in Section 4.4, nonlinearities that are symmetric around the origin are preferred because they tend to produce zero-mean inputs to the next layer (which is a desirable property). Empirically, we have observed that the *tanh* has better convergence properties.

### 5.4.2 Weight initialization

At initialization we want the weights to be small enough around the origin so that the activation function operates in its linear regime, where gradients are the largest. Other desirable properties, especially
for deep networks, are to conserve variance of the activation as well as variance of back-propagated gradients from layer to layer. This allows information to flow well upward and downward in the network and reduces discrepancies between layers. Under some assumptions, a compromise between these two constraints leads to the following initialization: \[
\text{uniform}[-\frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}}}+\text{fan}_{\text{out}}}, \frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}}}+\text{fan}_{\text{out}}}],
\]
for tanh and
\[
\text{uniform}[-4 \frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}}}+\text{fan}_{\text{out}}}, 4 \frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}}}+\text{fan}_{\text{out}}}],
\]
for sigmoid. Where \(\text{fan}_{\text{in}}\) is the number of inputs and \(\text{fan}_{\text{out}}\) the number of hidden units. For mathematical considerations please refer to [Xavier10].

5.4.3 Learning rate

There is a great deal of literature on choosing a good learning rate. The simplest solution is to simply have a constant rate. Rule of thumb: try several log-spaced values \((10^{-1}, 10^{-2}, \ldots)\) and narrow the (logarithmic) grid search to the region where you obtain the lowest validation error.

Decreasing the learning rate over time is sometimes a good idea. One simple rule for doing that is \(\mu(t) = \frac{\mu_0}{1 + d \times t}\), where \(\mu_0\) is the initial rate (chosen, perhaps, using the grid search technique explained above), \(d\) is a so-called “decrease constant” which controls the rate at which the learning rate decreases (typically, a smaller positive number, \(10^{-3}\) and smaller) and \(t\) is the epoch/stage.

Section 4.7 details procedures for choosing a learning rate for each parameter (weight) in our network and for choosing them adaptively based on the error of the classifier.

5.4.4 Number of hidden units

This hyper-parameter is very much dataset-dependent. Vaguely speaking, the more complicated the input distribution is, the more capacity the network will require to model it, and so the larger the number of hidden units that will be needed (note that the number of weights in a layer, perhaps a more direct measure of capacity, is \(D \times D_h\) (recall \(D\) is the number of inputs and \(D_h\) is the number of hidden units).

Unless we employ some regularization scheme (early stopping or L1/L2 penalties), a typical number of hidden units vs. generalization performance graph will be U-shaped.

5.4.5 Regularization parameter

Typical values to try for the L1/L2 regularization parameter \(\lambda\) are \(10^{-2}, 10^{-3}, \ldots\). In the framework that we described so far, optimizing this parameter will not lead to significantly better solutions, but is worth exploring nonetheless.
CHAPTER 6

Convolutional Neural Networks (LeNet)

Note: This section assumes the reader has already read through Classifying MNIST digits using Logistic Regression and Multilayer Perceptron. Additionally, it uses the following new Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, floatX, pool, conv2d, dimshuffle. If you intend to run the code on GPU also read GPU.

To run this example on a GPU, you need a good GPU. It needs at least 1GB of GPU RAM. More may be required if your monitor is connected to the GPU.

When the GPU is connected to the monitor, there is a limit of a few seconds for each GPU function call. This is needed as current GPUs can’t be used for the monitor while doing computation. Without this limit, the screen would freeze for too long and make it look as if the computer froze. This example hits this limit with medium-quality GPUs. When the GPU isn’t connected to a monitor, there is no time limit. You can lower the batch size to fix the time out problem.

Note: The code for this section is available for download here and the 3wolfmoon image

6.1 Motivation

Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. From Hubel and Wiesel’s early work on the cat’s visual cortex [Hubel68], we know the visual cortex contains a complex arrangement of cells. These cells are sensitive to small sub-regions of the visual field, called a receptive field. The sub-regions are tiled to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.
Additionally, two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

The animal visual cortex being the most powerful visual processing system in existence, it seems natural to emulate its behavior. Hence, many neurally-inspired models can be found in the literature. To name a few: the NeoCognitron \[\text{Fukushima}\], HMAX \[\text{Serre07}\] and LeNet-5 \[\text{LeCun98}\], which will be the focus of this tutorial.

### 6.2 Sparse Connectivity

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer \(m\) are from a subset of units in layer \(m-1\), units that have spatially contiguous receptive fields. We can illustrate this graphically as follows:

![Graph of Sparse Connectivity](image)

Imagine that layer \(m-1\) is the input retina. In the above figure, units in layer \(m\) have receptive fields of width 3 in the input retina and are thus only connected to 3 adjacent neurons in the retina layer. Units in layer \(m+1\) have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (5). Each unit is unresponsive to variations outside of its receptive field with respect to the retina. The architecture thus ensures that the learnt “filters” produce the strongest response to a spatially local input pattern.

However, as shown above, stacking many such layers leads to (non-linear) “filters” that become increasingly “global” (i.e. responsive to a larger region of pixel space). For example, the unit in hidden layer \(m+1\) can encode a non-linear feature of width 5 (in terms of pixel space).

### 6.3 Shared Weights

In addition, in CNNs, each filter \(h_i\) is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map.

In the above figure, we show 3 hidden units belonging to the same feature map. Weights of the same color are shared—constrained to be identical. Gradient descent can still be used to learn such shared parameters, with only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared.
Replicating units in this way allows for features to be detected \textit{regardless of their position in the visual field}. Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt. The constraints on the model enable CNNs to achieve better generalization on vision problems.

6.4 Details and Notation

A feature map is obtained by repeated application of a function across sub-regions of the entire image, in other words, by \textit{convolution} of the input image with a linear filter, adding a bias term and then applying a non-linear function. If we denote the k-th feature map at a given layer as $h^k$, whose filters are determined by the weights $W^k$ and bias $b_k$, then the feature map $h^k$ is obtained as follows (for $\tanh$ non-linearities):

$$h^k_{ij} = \tanh((W^k * x)_{ij} + b_k).$$

\textbf{Note:} Recall the following definition of convolution for a 1D signal. $o[n] = f[n] * g[n] = \sum_{u=-\infty}^{\infty} f[u]g[n-u]$. This can be extended to 2D as follows: $o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[m-u,n-v]$.

To form a richer representation of the data, each hidden layer is composed of \textit{multiple} feature maps, $\{h^{(k)}, k = 0..K\}$. The weights $W$ of a hidden layer can be represented in a 4D tensor containing elements for every combination of destination feature map, source feature map, source vertical position, and source horizontal position. The biases $b$ can be represented as a vector containing one element for every destination feature map. We illustrate this graphically as follows:

The figure shows two layers of a CNN. \textbf{Layer m-1} contains four feature maps. \textbf{Hidden layer m} contains two feature maps ($h^0$ and $h^1$). Pixels (neuron outputs) in $h^0$ and $h^1$ (outlined as blue and red squares) are computed from pixels of layer (m-1) which fall within their 2x2 receptive field in the layer below (shown as colored rectangles). Notice how the receptive field spans all four input feature maps. The weights $W^0$ and $W^1$ of $h^0$ and $h^1$ are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Putting it all together, $W^{kl}_{ij}$ denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i,j) of the l-th feature map of layer (m-1).
6.5 The Convolution Operator

ConvOp is the main workhorse for implementing a convolutional layer in Theano. ConvOp is used by `theano.tensor.signal.conv2d`, which takes two symbolic inputs:

- a 4D tensor corresponding to a mini-batch of input images. The shape of the tensor is as follows: [mini-batch size, number of input feature maps, image height, image width].
- a 4D tensor corresponding to the weight matrix $W$. The shape of the tensor is: [number of feature maps at layer m, number of feature maps at layer m-1, filter height, filter width]

Below is the Theano code for implementing a convolutional layer similar to the one of Figure 1. The input consists of 3 features maps (an RGB color image) of size 120x160. We use two convolutional filters with 9x9 receptive fields.

```python
import theano
from theano import tensor as T
from theano.tensor.nnet import conv2d

import numpy
rng = numpy.random.RandomState(23455)

# instantiate 4D tensor for input
input = T.tensor4(name='input')

# initialize shared variable for weights.
w_shp = (2, 3, 9, 9)
w_bound = numpy.sqrt(3 * 9 * 9)
W = theano.shared( numpy.asarray( rng.uniform(0, w_bound, w_shp), 
                          dtype=numpy.float32) )

# initialize shared variable for biases.
b_shp = (1, 2, 1, 1)
b = theano.shared( numpy.asarray( rng.uniform(-w_bound, w_bound, b_shp),
                          dtype=numpy.float32) )

# perform convolution
convout = conv2d(input, W)

# apply non-linearity
z = T.tanh(convout + b)
```

Fig. 6.1: **Figure 1**: example of a convolutional layer
low=-1.0 / w_bound,
       high=1.0 / w_bound,
       size=w_shp),
       dtype=input.dtype), name = 'W')

# initialize shared variable for bias (1D tensor) with random values
# IMPORTANT: biases are usually initialized to zero. However in this
# particular application, we simply apply the convolutional layer to
# an image without learning the parameters. We therefore initialize
# them to random values to "simulate" learning.

b_shp = (2,)
b = theano.shared(numpy.asarray(
    rng.uniform(low=-.5, high=.5, size=b_shp),
    dtype=input.dtype), name = 'b')

# build symbolic expression that computes the convolution of input with
# filters in W

conv_out = conv2d(input, W)

# build symbolic expression to add bias and apply activation function, i.e.
# produce neural net layer output

output = T.nnet.sigmoid(conv_out + b.dimshuffle('x', 0, 'x', 'x'))

# create theano function to compute filtered images
f = theano.function([input], output)

Let’s have a little bit of fun with this...

```python
import numpy
import pylab
from PIL import Image
```

6.5. The Convolution Operator
Deep Learning Tutorial, Release 0.1

```python
# open random image of dimensions 639x516
img = Image.open(open('doc/images/3wolfmoon.jpg'))
# dimensions are (height, width, channel)
img = numpy.asarray(img, dtype='float64') / 256.

# put image in 4D tensor of shape (1, 3, height, width)
img_ = img.transpose(2, 0, 1).reshape(1, 3, 639, 516)
filtered_img = f(img_)

# plot original image and first and second components of output
pylab.subplot(1, 3, 1); pylab.axis('off'); pylab.imshow(img)
pylab.gray();
# recall that the convOp output (filtered image) is actually a "minibatch",
# of size 1 here, so we take index 0 in the first dimension:
pylab.subplot(1, 3, 2); pylab.axis('off'); pylab.imshow(filtered_img[0, 0, :, :])
pylab.subplot(1, 3, 3); pylab.axis('off'); pylab.imshow(filtered_img[0, 1, :, :])
pylab.show()
```

This should generate the following output.

![Original Image](image)
![First Component](image)
![Second Component](image)

Notice that a randomly initialized filter acts very much like an edge detector!

Note that we use the same weight initialization formula as with the MLP. Weights are sampled randomly from a uniform distribution in the range [-1/fan-in, 1/fan-in], where fan-in is the number of inputs to a hidden unit. For MLPs, this was the number of units in the layer below. For CNNs however, we have to take into account the number of input feature maps and the size of the receptive fields.

### 6.6 MaxPooling

Another important concept of CNNs is max-pooling, which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.

Max-pooling is useful in vision for two reasons:

1. By eliminating non-maximal values, it reduces computation for upper layers.
2. It provides a form of translation invariance. Imagine cascading a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a
single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3x3 window, this jumps to 5/8.

Since it provides additional robustness to position, max-pooling is a “smart” way of reducing the dimensionality of intermediate representations.

Max-pooling is done in Theano by way of `theano.tensor.signal.pool.pool_2d`. This function takes as input an N dimensional tensor (where N >= 2) and a downscaling factor and performs max-pooling over the 2 trailing dimensions of the tensor.

An example is worth a thousand words:

```python
from theano.tensor.signal import pool

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = pool.pool_2d(input, maxpool_shape, ignore_border=True)
f = theano.function([input], pool_out)

invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)
print 'With ignore_border set to True:'
print 'invals[0, 0, :, :] =
', invals[0, 0, :, :]
print 'output[0, 0, :, :] =
', f(invals)[0, 0, :, :]

pool_out = pool.pool_2d(input, maxpool_shape, ignore_border=False)
f = theano.function([input], pool_out)
print 'With ignore_border set to False:
print 'invals[1, 0, :, :] =
', invals[1, 0, :, :]
print 'output[1, 0, :, :] =
', f(invals)[1, 0, :, :]
```

This should generate the following output:

With ignore_border set to True:
```
   [[ 4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01 1.46755891e-01]
   [ 4.1914514e-01 6.85219500e-01 2.04452250e-01 8.73875932e-02 7.3816734e-01]]
output[0, 0, :, :] =
[ [0.72032449 0.39674817]
 [0.6852195 0.87811744]]
```

With ignore_border set to False:
```
invals[1, 0, :, :] =
[[ 0.01936696 0.67883553 0.21162812 0.26554666 0.49157316]
 [ 0.05336255 0.57411761 0.14672857 0.58930554 0.69975836]
 [ 0.10233443 0.41405599 0.69440016 0.41417927 0.4995346]]
```
Note that compared to most Theano code, the `max_pool_2d` operation is a little *special*. It requires the downscaling factor `ds` (tuple of length 2 containing downscaling factors for image width and height) to be known at graph build time. This may change in the near future.

6.7 The Full Model: LeNet

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.

The lower-layers are composed of alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all feature maps at the layer below.

From an implementation point of view, this means lower-layers operate on 4D tensors. These are then flattened to a 2D matrix of rasterized feature maps, to be compatible with our previous MLP implementation.

Note: The term “convolution” could correspond to different mathematical operations:

1. `theano.tensor.nnet.conv2d`, which is the most common one in almost all of the recent published convolutional models. In this operation, each output feature map is connected to each input feature map by a different 2D filter, and its value is the sum of the individual convolution of all inputs through the corresponding filter.

2. The convolution used in the original LeNet model: In this work, each output feature map is only connected to a subset of input feature maps.

3. The convolution used in signal processing: `theano.tensor.signal.conv.conv2d`, which works only on single channel inputs.

Here, we use the first operation, so this models differ slightly from the original LeNet paper. One reason to use 2. would be to reduce the amount of computation needed, but modern hardware makes it as fast to have
the full connection pattern. Another reason would be to slightly reduce the number of free parameters, but we have other regularization techniques at our disposal.

6.8 Putting it All Together

We now have all we need to implement a LeNet model in Theano. We start with the LeNetConvPoolLayer class, which implements a \{convolution + max-pooling\} layer.

```python
class LeNetConvPoolLayer(object):
    '''Pool Layer of a convolutional network '''
    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2, 2)):
        '''Allocate a LeNetConvPoolLayer with shared variable internal parameters.

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.dtensor4
        :param input: symbolic image tensor, of shape image_shape

        :type filter_shape: tuple or list of length 4
        :param filter_shape: (number of filters, num input feature maps, filter height, filter width)

        :type image_shape: tuple or list of length 4
        :param image_shape: (batch size, num input feature maps, image height, image width)

        :type poolsize: tuple or list of length 2
        :param poolsize: the downsampling (pooling) factor (#rows, #cols)
        '''

        assert image_shape[1] == filter_shape[1]
        self.input = input

        # there are "num input feature maps * filter height * filter width"
        # inputs to each hidden unit
        fan_in = numpy.prod(filter_shape[1:])
        # each unit in the lower layer receives a gradient from:
        # "num output feature maps * filter height * filter width" /
        # pooling size
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:]) //
                   numpy.prod(poolsize))
        # initialize weights with random weights
        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                6.8. Putting it All Together 61
Notice that when initializing the weight values, the fan-in is determined by the size of the receptive fields and the number of input feature maps.

Finally, using the LogisticRegression class defined in *Classifying MNIST digits using Logistic Regression* and the HiddenLayer class defined in *Multilayer Perceptron*, we can instantiate the network as follows.
# to a 4D tensor, compatible with our LeNetConvPoolLayer
layer0_input = x.reshape((batch_size, 1, 28, 28))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1, 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 28, 28),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 12, 12),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 4 * 4,
    n_out=500,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer3.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer3.errors(y),
)
givens={
    x: test_set_x[index * batch_size: (index + 1) * batch_size],
    y: test_set_y[index * batch_size: (index + 1) * batch_size]
}

validate_model = theano.function(
    [index],
    layer3.errors(y),
    givens=
        {x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]}
)

# create a list of all model parameters to be fit by gradient descent
params = layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
# manually create an update rule for each model parameter. We thus
# create the updates list by automatically looping over all
# (params[i], grads[i]) pairs.
updates = [
    (param_i, param_i - learning_rate * grad_i)
    for param_i, grad_i in zip(params, grads)
]

train_model = theano.function(
    [index],
    cost,
    updates=updates,
    givens=
        {x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]}
)

We leave out the code that performs the actual training and early-stopping, since it is exactly the same as with an MLP. The interested reader can nevertheless access the code in the ‘code’ folder of DeepLearning-Tutorials.

### 6.9 Running the Code

The user can then run the code by calling:

```
python code/convolutional_mlp.py
```
The following output was obtained with the default parameters on a Core i7-2600K CPU clocked at 3.40GHz and using flags ‘floatX=float32’:

Optimization complete.
Best validation score of 0.910000 % obtained at iteration 17800, with test performance 0.920000 %
The code for file convolutional_mlp.py ran for 380.28m

Using a GeForce GTX 285, we obtained the following:

Optimization complete.
Best validation score of 0.910000 % obtained at iteration 15500, with test performance 0.930000 %
The code for file convolutional_mlp.py ran for 46.76m

And similarly on a GeForce GTX 480:

Optimization complete.
Best validation score of 0.910000 % obtained at iteration 16400, with test performance 0.930000 %
The code for file convolutional_mlp.py ran for 32.52m

Note that the discrepancies in validation and test error (as well as iteration count) are due to different implementations of the rounding mechanism in hardware. They can be safely ignored.

6.10 Tips and Tricks

6.10.1 Choosing Hyperparameters

CNNs are especially tricky to train, as they add even more hyper-parameters than a standard MLP. While the usual rules of thumb for learning rates and regularization constants still apply, the following should be kept in mind when optimizing CNNs.

Number of filters

When choosing the number of filters per layer, keep in mind that computing the activations of a single convolutional filter is much more expensive than with traditional MLPs!

Assume layer \((l-1)\) contains \(K^{l-1}\) feature maps and \(M \times N\) pixel positions (i.e., number of positions times number of feature maps), and there are \(K^l\) filters at layer \(l\) of shape \(m \times n\). Then computing a feature map (applying an \(m \times n\) filter at all \((M - m) \times (N - n)\) pixel positions where the filter can be applied) costs \((M - m) \times (N - n) \times m \times n \times K^{l-1}\). The total cost is \(K^l\) times that. Things may be more complicated if not all features at one level are connected to all features at the previous one.

For a standard MLP, the cost would only be \(K^l \times K^{l-1}\) where there are \(K^l\) different neurons at level \(l\). As such, the number of filters used in CNNs is typically much smaller than the number of hidden units in MLPs and depends on the size of the feature maps (itself a function of input image size and filter shapes).
Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more. In fact, to equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers. To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning). The number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task.

**Filter Shape**

Common filter shapes found in the literature vary greatly, usually based on the dataset. Best results on MNIST-sized images (28x28) are usually in the 5x5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12x12 or 15x15.

The trick is thus to find the right level of “granularity” (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset.

**Max Pooling Shape**

Typical values are 2x2 or no max-pooling. Very large input images may warrant 4x4 pooling in the lower-layers. Keep in mind however, that this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information.

**Tips**

If you want to try this model on a new dataset, here are a few tips that can help you get better results:

- Whitening the data (e.g. with PCA)
- Decay the learning rate in each epoch
CHAPTER 7

Denoising Autoencoders (dA)

Note: This section assumes the reader has already read through *Classifying MNIST digits using Logistic Regression* and *Multilayer Perceptron*. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX. If you intend to run the code on GPU also read GPU.

Note: The code for this section is available for download here.

The Denoising Autoencoder (dA) is an extension of a classical autoencoder and it was introduced as a building block for deep networks in [Vincent08]. We will start the tutorial with a short discussion on Autoencoders.

### 7.1 Autoencoders

See section 4.6 of [Bengio09] for an overview of auto-encoders. An autoencoder takes an input $x \in [0, 1]^d$ and first maps it (with an encoder) to a hidden representation $y \in [0, 1]^{d'}$ through a deterministic mapping, e.g.:

$$y = s(Wx + b)$$

Where $s$ is a non-linearity such as the sigmoid. The latent representation $y$, or **code** is then mapped back (with a decoder) into a **reconstruction** $z$ of the same shape as $x$. The mapping happens through a similar transformation, e.g.:

$$z = s(W'y + b')$$
Deep Learning Tutorial, Release 0.1

(Here, the prime symbol does not indicate matrix transposition.) z should be seen as a prediction of x, given the code y. Optionally, the weight matrix W′ of the reverse mapping may be constrained to be the transpose of the forward mapping: W′ = Wᵀ. This is referred to as tied weights. The parameters of this model (namely W, b, b' and, if one doesn’t use tied weights, also W′) are optimized such that the average reconstruction error is minimized.

The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code. The traditional squared error \( L(xz) = ||x - z||^2 \), can be used. If the input is interpreted as either bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

\[
L_H(x, z) = -\sum_{k=1}^{d} [x_k \log z_k + (1 - x_k) \log(1 - z_k)]
\]

The hope is that the code y is a distributed representation that captures the coordinates along the main factors of variation in the data. This is similar to the way the projection on principal components would capture the main factors of variation in the data. Indeed, if there is one linear hidden layer (the code) and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data. If the hidden layer is non-linear, the auto-encoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution. The departure from PCA becomes even more important when we consider stacking multiple encoders (and their corresponding decoders) when building a deep auto-encoder [Hinton06].

Because y is viewed as a lossy compression of x, it cannot be a good (small-loss) compression for all x. Optimization makes it a good compression for training examples, and hopefully for other inputs as well, but not for arbitrary inputs. That is the sense in which an auto-encoder generalizes: it gives low reconstruction error on test examples from the same distribution as the training examples, but generally high reconstruction error on samples randomly chosen from the input space.

We want to implement an auto-encoder using Theano, in the form of a class, that could be afterwards used in constructing a stacked autoencoder. The first step is to create shared variables for the parameters of the autoencoder W, b and b’. (Since we are using tied weights in this tutorial, Wᵀ will be used for W’):

```python
def __init__(self, numpy_rng, theano_rng=None, input=None, n_visible=784, n_hidden=500, W=None, bhid=None, bvis=None):
    """Initialize the dA class by specifying the number of visible units (the dimension d of the input), the number of hidden units (the dimension d’ of the latent or hidden space) and the corruption level. The constructor also receives symbolic variables for the input, weights and bias. Such a symbolic variables are useful when, for example the input is the result of some computations, or when weights are shared between
    ""
```
the dA and an MLP layer. When dealing with SdAs this always happens, the dA on layer 2 gets as input the output of the dA on layer 1, and the weights of the dA are used in the second stage of training to construct an MLP.

:type numpy_rng: numpy.random.RandomState
:param numpy_rng: number random generator used to generate weights

:type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
:param theano_rng: Theano random generator; if None is given one is generated based on a seed drawn from `rng`

:type input: theano.tensor.TensorType
:param input: a symbolic description of the input or None for standalone dA

:type n_visible: int
:param n_visible: number of visible units

:type n_hidden: int
:param n_hidden: number of hidden units

:type W: theano.tensor.TensorType
:param W: Theano variable pointing to a set of weights that should be shared belong the dA and another architecture; if dA should be standalone set this to None

:type bhid: theano.tensor.TensorType
:param bhid: Theano variable pointing to a set of biases values (for hidden units) that should be shared belong dA and another architecture; if dA should be standalone set this to None

:type bvis: theano.tensor.TensorType
:param bvis: Theano variable pointing to a set of biases values (for visible units) that should be shared belong dA and another architecture; if dA should be standalone set this to None

""

self.n_visible = n_visible
self.n_hidden = n_hidden

# create a Theano random generator that gives symbolic random values
if not theano_rng:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

# note : W' was written as `W_prime` and b' as `b_prime`
if not W:
    # W is initialized with `initial_W` which is uniformly sampled
    # from -4*sqrt(6./(n_visible+n_hidden)) and
    # 4*sqrt(6.)/(n_hidden+n_visible)) the output of uniform if
    # converted using asarray to dtype

7.1. Autoencoders
# theano.config.floatX so that the code is runnable on GPU
initial_W = numpy.asarray(
    numpy_rng.uniform(
        low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
        high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
        size=(n_visible, n_hidden)
    ),
    dtype=theano.config.floatX
)
W = theano.shared(value=initial_W, name='W', borrow=True)

if not bvis:
    bvis = theano.shared(
        value=numpy.zeros(
            n_visible,
            dtype=theano.config.floatX
        ),
        borrow=True
    )

if not bhid:
    bhid = theano.shared(
        value=numpy.zeros(
            n_hidden,
            dtype=theano.config.floatX
        ),
        name='b',
        borrow=True
    )

self.W = W
# b corresponds to the bias of the hidden
self.b = bhid
# b_prime corresponds to the bias of the visible
self.b_prime = bvis
# tied weights, therefore W_prime is W transpose
self.W_prime = self.W.T
self.theano_rng = theano_rng
# if no input is given, generate a variable representing the input
if input is None:
    # we use a matrix because we expect a minibatch of several
    # examples, each example being a row
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]

Note that we pass the symbolic input to the autoencoder as a parameter. This is so that we can concatenate layers of autoencoders to form a deep network: the symbolic output (the y above) of layer k will be the symbolic input of layer k + 1.

Now we can express the computation of the latent representation and of the reconstructed signal:
def get_hidden_values(self, input):
    """ Computes the values of the hidden layer """
    return T.nnet.sigmoid(T.dot(input, self.W) + self.b)

def get_reconstructed_input(self, hidden):
    """Computes the reconstructed input given the values of the hidden layer
    """
    return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)

And using these functions we can compute the cost and the updates of one stochastic gradient descent step:

def get_cost_updates(self, corruption_level, learning_rate):
    """ This function computes the cost and the updates for one training step of the dA """

tilde_x = self.get_corrupted_input(self.x, corruption_level)
y = self.get_hidden_values(tilde_x)
z = self.get_reconstructed_input(y)
    # note : we sum over the size of a datapoint; if we are using minibatches, L will be a vector, with one entry per example in minibatch
L = - T.sum(self.x * T.log(z) + (1 - self.x) * T.log(1 - z), axis=1)
    # note : L is now a vector, where each element is the cross-entropy cost of the reconstruction of the corresponding example of the minibatch. We need to compute the average of all these to get the cost of the minibatch
    cost = T.mean(L)

    # compute the gradients of the cost of the `dA` with respect to its parameters
    gparams = T.grad(cost, self.params)
    # generate the list of updates
    updates = [ (param, param - learning_rate * gparam)
                for param, gparam in zip(self.params, gparams) ]

    return (cost, updates)

We can now define a function that applied iteratively will update the parameters $W$, $b$ and $b_\prime$ such that the reconstruction cost is approximately minimized.

da = dA(
    numpy_rng=rng,
    theano_rng=theano_rng,
    input=x,
    n_visible=28 * 28,
    n_hidden=500
)

7.1. Autoencoders
```python
cost, updates = da.get_cost_updates(
    corruption_level=0.,
    learning_rate=learning_rate
)

train_da = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size]
    }
)

start_time = timeit.default_timer()

############
# TRAINING #
############

# go through training epochs
for epoch in range(training_epochs):
    # go through training set
    c = []
    for batch_index in range(n_train_batches):
        c.append(train_da(batch_index))

    print('Training epoch %d, cost ' % epoch, numpy.mean(c, dtype='float64'))

end_time = timeit.default_timer()

training_time = (end_time - start_time)

print(('The no corruption code for file ' + os.path.split(__file__)[1] + ' ran for %.2fm' % ((training_time) / 60.)), file=sys.stderr)

tile_raster_images(X=da.W.get_value(borrow=True).T,
    img_shape=(28, 28), tile_shape=(10, 10),
    tile_spacing=(1, 1)))

image.save('filters_corruption_0.png')

# start-snippet-3

rng = numpy.random.RandomState(123)
theano_rng = RandomStreams(rng.randint(2 ** 30))
```

---

72 Chapter 7. Denoising Autoencoders (dA)
da = dA(
    numpy_rng=rng,
    theano_rng=theano_rng,
    input=x,
    n_visible=28 * 28,
    n_hidden=500
)

cost, updates = da.get_cost_updates(
    corruption_level=0.3,
    learning_rate=learning_rate
)

train_da = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size]
    }
)

start_time = timeit.default_timer()

# TRAINING

# go through training epochs
for epoch in range(training_epochs):
    # go through training set
    c = []
    for batch_index in range(n_train_batches):
        c.append(train_da(batch_index))

    print('Training epoch %d, cost ' % epoch, numpy.mean(c, dtype='float64'))

end_time = timeit.default_timer()

training_time = (end_time - start_time)

print(('The 30% corruption code for file ' + os.path.split(__file__)[1] + ' ran for %.2f m' % (training_time / 60)), file=sys.stderr)

# start-snippet-4

tile_raster_images = image_fromarray(tile_raster_images(
    X=da.W.get_value(borrow=True).T,
    img_shape=(28, 28), tile_shape=(10, 10),
    tile_spacing=(1, 1)))
image.save('filters_corruption_30.png')

7.1. Autoencoders
If there is no constraint besides minimizing the reconstruction error, one might expect an auto-encoder with $n$ inputs and an encoding of dimension $n$ (or greater) to learn the identity function, merely mapping an input to its copy. Such an autoencoder would not differentiate test examples (from the training distribution) from other input configurations.

Surprisingly, experiments reported in [Bengio07] suggest that, in practice, when trained with stochastic gradient descent, non-linear auto-encoders with more hidden units than inputs (called overcomplete) yield useful representations. (Here, “useful” means that a network taking the encoding as input has low classification error.)

A simple explanation is that stochastic gradient descent with early stopping is similar to an L2 regularization of the parameters. To achieve perfect reconstruction of continuous inputs, a one-hidden layer auto-encoder with non-linear hidden units (exactly like in the above code) needs very small weights in the first (encoding) layer, to bring the non-linearity of the hidden units into their linear regime, and very large weights in the second (decoding) layer. With binary inputs, very large weights are also needed to completely minimize the reconstruction error. Since the implicit or explicit regularization makes it difficult to reach large-weight solutions, the optimization algorithm finds encodings which only work well for examples similar to those in the training set, which is what we want. It means that the representation is exploiting statistical regularities present in the training set, rather than merely learning to replicate the input.

There are other ways by which an auto-encoder with more hidden units than inputs could be prevented from learning the identity function, capturing something useful about the input in its hidden representation. One is the addition of sparsity (forcing many of the hidden units to be zero or near-zero). Sparsity has been exploited very successfully by many [Ranzato07] [Lee08]. Another is to add randomness in the transformation from input to reconstruction. This technique is used in Restricted Boltzmann Machines (discussed later in Restricted Boltzmann Machines (RBM)), as well as in Denoising Auto-Encoders, discussed below.

### 7.2 Denoising Autoencoders

The idea behind denoising autoencoders is simple. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, we train the autoencoder to reconstruct the input from a corrupted version of it.

The denoising auto-encoder is a stochastic version of the auto-encoder. Intuitively, a denoising auto-encoder does two things: try to encode the input (preserve the information about the input), and try to undo the effect of a corruption process stochastically applied to the input of the auto-encoder. The latter can only be done by capturing the statistical dependencies between the inputs. The denoising auto-encoder can be understood from different perspectives (the manifold learning perspective, stochastic operator perspective, bottom-up – information theoretic perspective, top-down – generative model perspective), all of which are explained in [Vincent08]. See also section 7.2 of [Bengio09] for an overview of auto-encoders.
In [Vincent08], the stochastic corruption process randomly sets some of the inputs (as many as half of them) to zero. Hence the denoising auto-encoder is trying to predict the corrupted (i.e. missing) values from the uncorrupted (i.e., non-missing) values, for randomly selected subsets of missing patterns. Note how being able to predict any subset of variables from the rest is a sufficient condition for completely capturing the joint distribution between a set of variables (this is how Gibbs sampling works).

To convert the autoencoder class into a denoising autoencoder class, all we need to do is to add a stochastic corruption step operating on the input. The input can be corrupted in many ways, but in this tutorial we will stick to the original corruption mechanism of randomly masking entries of the input by making them zero. The code below does just that:

```python
def get_corrupted_input(self, input, corruption_level):
    '''This function keeps `1-corruption_level` entries of the inputs
    the same and zero-out randomly selected subset of size `corruption_level`.
    Note : first argument of theano.rng.binomial is the shape(size) of
    random numbers that it should produce
    second argument is the number of trials
    third argument is the probability of success of any trial
    this will produce an array of 0s and 1s where 1 has a
    probability of 1 - `corruption_level` and 0 with
    `corruption_level`.

    The binomial function return int64 data type by
default. int64 multiplicated by the input
type(floatX) always return float64. To keep all data
in floatX when floatX is float32, we set the dtype of
the binomial to floatX. As in our case the value of
the binomial is always 0 or 1, this don't change the
result. This is needed to allow the gpu to work
correctly as it only support float32 for now.

    '''
    return self.theano_rng.binomial(size=input.shape, n=1,
                                     p=1 - corruption_level,
                                     dtype=theano.config.floatX) * input
```

In the stacked autoencoder class (Stacked Autoencoders) the weights of the dA class have to be shared with those of a corresponding sigmoid layer. For this reason, the constructor of the dA also gets Theano variables pointing to the shared parameters. If those parameters are left to `None`, new ones will be constructed.

The final denoising autoencoder class becomes:

```python
class dA(object):
    '''Denoising Auto-Encoder class (dA)

    A denoising autoencoders tries to reconstruct the input from a corrupted
    version of it by projecting it first in a latent space and reprojecting
    it afterwards back in the input space. Please refer to Vincent et al.,2008
    for more details. If x is the input then equation (1) computes a partially
    destroyed version of x by means of a stochastic mapping q_D. Equation (2)
```
computes the projection of the input into the latent space. Equation (3) computes the reconstruction of the input, while equation (4) computes the reconstruction error.

.. math::

\tilde{x} \sim q_D(\tilde{x}|x) \quad (1) \\
y = s(W \tilde{x} + b) \quad (2) \\
x = s(W'y + b') \quad (3) \\
L(x,z) = -\sum_{k=1}^{d} [x_k \log z_k + (1-x_k) \log(1-z_k)] \quad (4)

````

def __init__(self, 
numpy_rng, 
theano_rng=None, 
input=None, 
n_visible=784, 
n_hidden=500, 
W=None, 
bhid=None, 
bvis=None 
):
    ""
    Initialize the dA class by specifying the number of visible units (the dimension $d$ of the input), the number of hidden units (the dimension $d'$ of the latent or hidden space) and the corruption level. The constructor also receives symbolic variables for the input, weights, and bias. Such a symbolic variables are useful when, for example the input is the result of some computations, or when weights are shared between the dA and an MLP layer. When dealing with SdAs this always happens, the dA on layer 2 gets as input the output of the dA on layer 1, and the weights of the dA are used in the second stage of training to construct an MLP.

:type numpy_rng: numpy.random.RandomState
:param numpy_rng: number random generator used to generate weights

:type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
:param theano_rng: Theano random generator; if None is given one is generated based on a seed drawn from `rng`.

:type input: theano.tensor.TensorType
:param input: a symbolic description of the input or None for standalone dA

:type n_visible: int
:param n_visible: number of visible units
````
```python
:type n_hidden: int
:param n_hidden: number of hidden units

:type W: theano.tensor.TensorType
:param W: Theano variable pointing to a set of weights that should be
shared belong the dA and another architecture; if dA should be standalone set this to None

:type bhid: theano.tensor.TensorType
:param bhid: Theano variable pointing to a set of biases values (for
hidden units) that should be shared belong dA and another
architecture; if dA should be standalone set this to None

:type bvis: theano.tensor.TensorType
:param bvis: Theano variable pointing to a set of biases values (for
visible units) that should be shared belong dA and another
architecture; if dA should be standalone set this to None

```
if not bhid:
    bhid = theano.shared(
        value=numpy.zeros(
            n_hidden,
            dtype=theano.config.floatX
        ),
        name='b',
        borrow=True
    )

self.W = W
# b corresponds to the bias of the hidden
self.b = bhid
# b_prime corresponds to the bias of the visible
self.b_prime = bvis
# tied weights, therefore W_prime is W transpose
self.W_prime = self.W.T
self.theano_rng = theano_rng
# if no input is given, generate a variable representing the input
if input is None:
    # we use a matrix because we expect a minibatch of several
    # examples, each example being a row
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]

def get_corrupted_input(self, input, corruption_level):
    """This function keeps `1-corruption_level` entries of the inputs
    the same and zero-out randomly selected subset of size `corruption_"""
    "level`.
    Note: first argument of theano.rng.binomial is the shape(size) of
    random numbers that it should produce
    second argument is the number of trials
    third argument is the probability of success of any trial
    this will produce an array of 0s and 1s where 1 has a
    probability of 1 - `corruption_level` and 0 with
    `corruption_level`

    The binomial function return int64 data type by default. int64 multiplied by the input
type(floatX) always return float64. To keep all data
in floatX when floatX is float32, we set the dtype of
the binomial to floatX. As in our case the value of
the binomial is always 0 or 1, this don't change the
result. This is needed to allow the gpu to work
correctly as it only support float32 for now.
def get_hidden_values(self, input):
    """ Computes the values of the hidden layer """
    return T.nnet.sigmoid(T.dot(input, self.W) + self.b)

def get_reconstructed_input(self, hidden):
    """Computes the reconstructed input given the values of the hidden layer
    """
    return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)

def get_cost_updates(self, corruption_level, learning_rate):
    """ This function computes the cost and the updates for one training step of the dA """
    tilde_x = self.get_corrupted_input(self.x, corruption_level)
y = self.get_hidden_values(tilde_x)
z = self.get_reconstructed_input(y)
    # note : we sum over the size of a datapoint; if we are using
    # minibatches, L will be a vector, with one entry per
    # example in minibatch
    L = - T.sum(self.x * T.log(z) + (1 - self.x) * T.log(1 - z), axis=1)
    # note : L is now a vector, where each element is the
    # cross-entropy cost of the reconstruction of the
    # corresponding example of the minibatch. We need to
    # compute the average of all these to get the cost of
    # the minibatch
    cost = T.mean(L)
    # compute the gradients of the cost of the `dA` with respect
    # to its parameters
    gparams = T.grad(cost, self.params)
    # generate the list of updates
    updates = [
        (param, param - learning_rate * gparam)
        for param, gparam in zip(self.params, gparams)
    ]
    return (cost, updates)

7.3 Putting it All Together

It is easy now to construct an instance of our dA class and train it.

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch

7.3. Putting it All Together 79
\[ x = \text{T.matrix('x')} \]  # the data is presented as rasterized images

```python
rng = \text{numpy.random.RandomState(123)}
theano_rng = \text{RandomStreams(rng.randint(2 ** 30))}

da = \text{dA(}
    \text{numpy_rng=rng,}
    \text{theano_rng=theano_rng,}
    \text{input=x,}
    \text{n_visible=28 * 28,}
    \text{n_hidden=500}
\}

cost, updates = da.get_cost_updates(
    \text{corruption_level=0.3,}
    \text{learning_rate=learning_rate}
)

train_da = \text{theano.function(}
    \text{[index],}
    \text{cost,}
    \text{updates=updates,}
    \text{givens={}
        \text{x: train_set_x[index * batch_size: (index + 1) * batch_size]}
    \}}
)

\text{start_time = timeit.default_timer()}

\text{for epoch in range(training_epochs):}
    \text{# go through training set}
    \text{c = [ ]}
    \text{for batch_index in range(n_train_batches):}
        \text{c.append(train_da(batch_index))}

        \text{print('Training epoch %d, cost ' % epoch, numpy.mean(c, dtype='float64'))}

\text{end_time = timeit.default_timer()}

\text{training_time = (end_time - start_time)}

\text{print('The 30\% corruption code for file ' + os.path.split(__file__)[1] + '...')}
```
In order to get a feeling of what the network learned we are going to plot the filters (defined by the weight matrix). Bear in mind, however, that this does not provide the entire story, since we neglect the biases and plot the weights up to a multiplicative constant (weights are converted to values between 0 and 1).

To plot our filters we will need the help of \texttt{tile\_raster\_images} (see \textit{Plotting Samples and Filters}) so we urge the reader to study it. Also using the help of the Python Image Library, the following lines of code will save the filters as an image:

```python
image = Image.fromarray(tile_raster_images(
    X=da.W.get_value(borrow=True).T,
    img_shape=(28, 28), tile_shape=(10, 10),
    tile_spacing=(1, 1)))
image.save('filters_corruption_30.png')
```

### 7.4 Running the Code

To run the code:

```bash
python dA.py
```

The resulted filters when we do not use any noise are:

![Filters without noise](image)

The filters for 30 percent noise:
Note: This section assumes you have already read through Classifying MNIST digits using Logistic Regression and Multilayer Perceptron. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX. If you intend to run the code on GPU also read GPU.

The Stacked Denoising Autoencoder (SdA) is an extension of the stacked autoencoder [Bengio07] and it was introduced in [Vincent08].

This tutorial builds on the previous tutorial Denoising Autoencoders. Especially if you do not have experience with autoencoders, we recommend reading it before going any further.

8.1 Stacked Autoencoders

Denoising autoencoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising autoencoder found on the layer below as input to the current layer. The **unsupervised pre-training** of such an architecture is done one layer at a time. Each layer is trained as a denoising autoencoder by minimizing the error in reconstructing its input (which is the output code of the previous layer). Once the first $k$ layers are trained, we can train the $k + 1$-th layer because we can now compute the code or latent representation from the layer below.

Once all layers are pre-trained, the network goes through a second stage of training called **fine-tuning**. Here we consider **supervised fine-tuning** where we want to minimize prediction error on a supervised task. For this, we first add a logistic regression layer on top of the network (more precisely on the output code of the
output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training. (See the *Multilayer Perceptron* for details on the multilayer perceptron.)

This can be easily implemented in Theano, using the class defined previously for a denoising autoencoder. We can see the stacked denoising autoencoder as having two facades: a list of autoencoders, and an MLP. During pre-training we use the first facade, i.e., we treat our model as a list of autoencoders, and train each autoencoder separately. In the second stage of training, we use the second facade. These two facades are linked because:

- the autoencoders and the sigmoid layers of the MLP share parameters, and
- the latent representations computed by intermediate layers of the MLP are fed as input to the autoencoders.

class SdA(object):
    """Stacked denoising auto-encoder class (SdA)

    A stacked denoising autoencoder model is obtained by stacking several
dAs. The hidden layer of the dA at layer `i` becomes the input of
the dA at layer `i+1`. The first layer dA gets as input the input of
the SdA, and the hidden layer of the last dA represents the output.
Note that after pretraining, the SdA is dealt with as a normal MLP,
the dAs are only used to initialize the weights.
    """

def __init__(
    self,
    numpy_rng,
    theano_rng=None,
    n_ins=784,
    hidden_layers_sizes=[500, 500],
    n_outs=10,
    corruption_levels=[0.1, 0.1]
):  
    """ This class is made to support a variable number of layers. 

    :type numpy_rng: numpy.random.RandomState
    :param numpy_rng: numpy random number generator used to draw initial
weights

    :type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
    :param theano_rng: Theano random generator; if None is given one is
    generated based on a seed drawn from `rng`

    :type n_ins: int
    :param n_ins: dimension of the input to the sdA

    :type hidden_layers_sizes: list of ints
    :param hidden_layers_sizes: intermediate layers size, must contain
    at least one value

    :type n_outs: int
    :param n_outs: dimension of the output of the network

84 Chapter 8. Stacked Denoising Autoencoders (SdA)
self.sigmoid_layers will store the sigmoid layers of the MLP facade, while self.dA_layers will store the denoising autoencoder associated with the layers of the MLP.

Next, we construct n_layers sigmoid layers and n_layers denoising autoencoders, where n_layers is the depth of our model. We use the HiddenLayer class introduced in Multilayer Perceptron, with one modification: we replace the tanh non-linearity with the logistic function \( s(x) = \frac{1}{1+e^{-x}} \). We link the sigmoid layers to form an MLP, and construct the denoising autoencoders such that each shares the weight matrix and the bias of its encoding part with its corresponding sigmoid layer.

```python
for i in range(self.n_layers):
    # construct the sigmoidal layer
    
    # the size of the input is either the number of hidden units of
    # the layer below or the input size if we are on the first layer
    if i == 0:
        input_size = n_ins
    else:
        input_size = hidden_layers_sizes[i-1]

    # the input to this layer is either the activation of the hidden
    # layer below or the input of the SdA if you are on the first
    # layer
    if i == 0:
        layer_input = self.x
    else:
        layer_input = self.sigmoid_layers[-1].output

    sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                input=layer_input,
                                n_in=input_size,
                                n_out=hidden_layers_sizes[i],
                                activation=T.nnet.sigmoid)

    # add the layer to our list of layers
```

8.1. Stacked Autoencoders
self.sigmoid_layers.append(sigmoid_layer)
# its arguably a philosophical question...
# but we are going to only declare that the parameters of the
# sigmoid_layers are parameters of the StackedDA
# the visible biases in the dA are parameters of those
dA, but not the SdA
self.params.extend(sigmoid_layer.params)

# Construct a denoising autoencoder that shared weights with this
# layer
dA_layer = dA(numpy_rng=numpy_rng,
    theano_rng=theano_rng,
    input=layer_input,
    n_visible=input_size,
    n_hidden=hidden_layers_sizes[i],
    W=sigmoid_layer.W,
    bhid=sigmoid_layer.b)
self.dA_layers.append(dA_layer)

All we need now is to add a logistic layer on top of the sigmoid layers such that we have an MLP. We will
use the LogisticRegression class introduced in "Classifying MNIST digits using Logistic Regression."

# We now need to add a logistic layer on top of the MLP
self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs
)

self.params.extend(self.logLayer.params)
# construct a function that implements one step of finetuning

# compute the cost for second phase of training,
# defined as the negative log likelihood
self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
# compute the gradients with respect to the model parameters
# symbolic variable that points to the number of errors made on the
# minibatch given by self.x and self.y
self.errors = self.logLayer.errors(self.y)

The SdA class also provides a method that generates training functions for the denoising autoencoders in its
layers. They are returned as a list, where element i is a function that implements one step of training the dA
corresponding to layer i.

def pretraining_functions(self, train_set_x, batch_size):
    ''' Generates a list of functions, each of them implementing one
    step in training the dA corresponding to the layer with same index.
    The function will require as input the minibatch index, and to train
    a dA you just need to iterate, calling the corresponding function on
    all minibatch indexes.
To be able to change the corruption level or the learning rate during training, we associate Theano variables with them.

```python
corruption_level = T.scalar('corruption')  # % of corruption to use
learning_rate = T.scalar('lr')  # learning rate to use
# beginning of a batch, given `index`
batch_begin = index * batch_size
# ending of a batch given `index`
batch_end = batch_begin + batch_size

pretrain_fns = []
for dA in self.dA_layers:
    # get the cost and the updates list
    cost, updates = dA.get_cost_updates(corruption_level,
                                         learning_rate)
    # compile the theano function
    fn = theano.function(
        inputs=[
            index,
            theano.In(corruption_level, value=0.2),
            theano.In(learning_rate, value=0.1)
        ],
        outputs=cost,
        updates=updates,
        givens={
            self.x: train_set_x[batch_begin: batch_end]
        }
    )
    # append `fn` to the list of functions
    pretrain_fns.append(fn)

return pretrain_fns
```

Now any function `pretrain_fns[i]` takes as arguments `index` and optionally `corruption`—the corruption level or `lr`—the learning rate. Note that the names of the parameters are the names given to the Theano variables when they are constructed, not the names of the Python variables (`learning_rate` or

8.1. Stacked Autoencoders
corruption_level). Keep this in mind when working with Theano.

In the same fashion we build a method for constructing the functions required during finetuning (\texttt{train_fn}, \texttt{valid_score} and \texttt{test_score}).

```python
def build_finetune_functions(self, datasets, batch_size, learning_rate):
    '''Generates a function `train` that implements one step of
    finetuning, a function `validate` that computes the error on
    a batch from the validation set, and a function `test` that
    computes the error on a batch from the testing set

    :type datasets: list of pairs of theano.tensor.TensorType
    :param datasets: It is a list that contain all the datasets;
        the has to contain three pairs, `train`,
        `valid`, `test` in this order, where each pair
        is formed of two Theano variables, one for the
data points, the other for the labels

    :type batch_size: int
    :param batch_size: size of a minibatch

    :type learning_rate: float
    :param learning_rate: learning rate used during finetune stage
    '''

    (train_set_x, train_set_y) = datasets[0]
    (valid_set_x, valid_set_y) = datasets[1]
    (test_set_x, test_set_y) = datasets[2]

    # compute number of minibatches for training, validation and testing
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
    n_valid_batches //= batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0]
    n_test_batches //= batch_size

    index = T.lscalar('index')  # index to a [mini]batch

    # compute the gradients with respect to the model parameters
    gparams = T.grad(self.finetune_cost, self.params)

    # compute list of fine-tuning updates
    updates = [
        (param, param - gparam * learning_rate)
        for param, gparam in zip(self.params, gparams)
    ]

    train_fn = theano.function(
        inputs=[index],
        outputs=self.finetune_cost,
        updates=updates,
        givens={
            self.x: train_set_x[
                index * batch_size: (index + 1) * batch_size
            ],
```
def valid_score():
    return [valid_score_i(i) for i in range(n_valid_batches)]

# Create a function that scans the entire test set
def test_score():
    return [test_score_i(i) for i in range(n_test_batches)]

return train_fn, valid_score, test_score

Note that valid_score and test_score are not Theano functions, but rather Python functions that loop over the entire validation set and the entire test set, respectively, producing a list of the losses over these sets.
8.2 Putting it all together

The few lines of code below construct the stacked denoising autoencoder:

```python
numpy_rng = numpy.random.RandomState(89677)
print('... building the model')
# construct the stacked denoising autoencoder class
sda = SdA(
    numpy_rng=numpy_rng,
    n_ins=28 * 28,
    hidden_layers_sizes=[1000, 1000, 1000],
    n_outs=10
)
```

There are two stages of training for this network: layer-wise pre-training followed by fine-tuning.

For the pre-training stage, we will loop over all the layers of the network. For each layer we will use the compiled Theano function that implements a SGD step towards optimizing the weights for reducing the reconstruction cost of that layer. This function will be applied to the training set for a fixed number of epochs given by `pretraining_epochs`.

```python
pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
                                           batch_size=batch_size)

print('... pre-training the model')
start_time = timeit.default_timer()
## Pre-train layer-wise
corruption_levels = [.1, .2, .3]
for i in range(sda.n_layers):
    # go through pretraining epochs
    for epoch in range(pretraining_epochs):
        # go through the training set
        c = []
        for batch_index in range(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
                                         corruption=corruption_levels[i],
                                         lr=pretrain_lr))
        print('Pre-training layer %i, epoch %d, cost %f' % (i, epoch,
                                                             numpy.mean(c, dtype='float64')))
    end_time = timeit.default_timer()

    print('The pretraining code for file ' +
          os.path.split(__file__)[1] +
          ' ran for %.2fm' % ((end_time - start_time) / 60.))
```

The fine-tuning loop is very similar to the one in the *Multilayer Perceptron*. The only difference is that it uses the functions given by `build_finetune_functions`.
8.3 Running the Code

The user can run the code by calling:

```
python code/SdA.py
```

By default the code runs 15 pre-training epochs for each layer, with a batch size of 1. The corruption levels are 0.1 for the first layer, 0.2 for the second, and 0.3 for the third. The pretraining learning rate is 0.001 and the finetuning learning rate is 0.1. Pre-training takes 585.01 minutes, with an average of 13 minutes per epoch. Fine-tuning is completed after 36 epochs in 444.2 minutes, with an average of 12.34 minutes per epoch. The final validation score is 1.39% with a testing score of 1.3%. These results were obtained on a machine with an Intel Xeon E5430 @ 2.66GHz CPU, with a single-threaded GotoBLAS.

8.4 Tips and Tricks

One way to improve the running time of your code (assuming you have sufficient memory available), is to compute how the network, up to layer $k - 1$, transforms your data. Namely, you start by training your first layer $dA$. Once it is trained, you can compute the hidden units values for every datapoint in your dataset and store this as a new dataset that you will use to train the $dA$ corresponding to layer 2. Once you have trained the $dA$ for layer 2, you compute, in a similar fashion, the dataset for layer 3 and so on. You can see now, that at this point, the $dAs$ are trained individually, and they just provide (one to the other) a non-linear transformation of the input. Once all $dAs$ are trained, you can start fine-tuning the model.


Restricted Boltzmann Machines (RBM)

Note: This section assumes the reader has already read through *Classifying MNIST digits using Logistic Regression* and *Multilayer Perceptron*. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX and scan. If you intend to run the code on GPU also read GPU.

Note: The code for this section is available for download here.

9.1 Energy-Based Models (EBM)

Energy-based models associate a scalar energy to each configuration of the variables of interest. Learning corresponds to modifying that energy function so that its shape has desirable properties. For example, we would like plausible or desirable configurations to have low energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows:

\[ p(x) = \frac{e^{-E(x)}}{Z} \]  

(9.1)

The normalizing factor \( Z \) is called the partition function by analogy with physical systems.

\[ Z = \sum_x e^{-E(x)} \]

An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. As for the logistic regression we will first define the log-likelihood and
then the loss function as being the negative log-likelihood.

\[
\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log p(x^{(i)})
\]

\[
\ell(\theta, \mathcal{D}) = -\mathcal{L}(\theta, \mathcal{D})
\]

using the stochastic gradient \(-\frac{\partial \log p(x^{(i)})}{\partial \theta}\), where \(\theta\) are the parameters of the model.

**EBMs with Hidden Units**

In many cases of interest, we do not observe the example \(x\) fully, or we want to introduce some non-observed variables to increase the expressive power of the model. So we consider an observed part (still denoted \(x\) here) and a hidden part \(h\). We can then write:

\[
P(x) = \sum_h P(x, h) = \sum_h \frac{e^{-E(x,h)}}{Z}.
\]

(9.2)

In such cases, to map this formulation to one similar to Eq. (9.1), we introduce the notation (inspired from physics) of free energy, defined as follows:

\[
\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)}
\]

(9.3)

which allows us to write,

\[
P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.
\]

The data negative log-likelihood gradient then has a particularly interesting form.

\[
-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}.
\]

(9.4)

Notice that the above gradient contains two terms, which are referred to as the positive and negative phase. The terms positive and negative do not refer to the sign of each term in the equation, but rather reflect their effect on the probability density defined by the model. The first term increases the probability of training data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model.

It is usually difficult to determine this gradient analytically, as it involves the computation of \(E_F\left(\frac{\partial \mathcal{F}(x)}{\partial \theta}\right)\). This is nothing less than an expectation over all possible configurations of the input \(x\) (under the distribution \(P\) formed by the model)!

The first step in making this computation tractable is to estimate the expectation using a fixed number of model samples. Samples used to estimate the negative phase gradient are referred to as negative particles, which are denoted as \(\mathcal{N}\). The gradient can then be written as:

\[
-\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|} \sum_{\tilde{x} \in \mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}.
\]

(9.5)

where we would ideally like elements \(\tilde{x}\) of \(\mathcal{N}\) to be sampled according to \(P\) (i.e. we are doing Monte-Carlo). With the above formula, we almost have a practical, stochastic algorithm for learning an EBM. The only missing ingredient is how to extract these negative particles \(\mathcal{N}\). While the statistical literature abounds with sampling methods, Markov Chain Monte Carlo methods are especially well suited for models such as the Restricted Boltzmann Machines (RBM), a specific type of EBM.
9.2 Restricted Boltzmann Machines (RBM)

Boltzmann Machines (BMs) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown below.

The energy function \( E(v, h) \) of an RBM is defined as:

\[
E(v, h) = -b'v - c'h - h'Wv
\]  

(9.6)

where \( W \) represents the weights connecting hidden and visible units and \( b, c \) are the offsets of the visible and hidden layers respectively.

This translates directly to the following free energy formula:

\[
\mathcal{F}(v) = -b'v - \sum_i \log \sum_h e^{h_i(c_i + W_i v)}.
\]

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

\[
p(h|v) = \prod_i p(h_i|v)
\]

\[
p(v|h) = \prod_j p(v_j|h).
\]

**RBMs with binary units**

In the commonly studied case of using binary units (where \( v_j \) and \( h_i \in \{0, 1\} \)), we obtain from Eq. (9.6) and (9.2), a probabilistic version of the usual neuron activation function:

\[
P(h_i = 1|v) = \text{sigm}(c_i + W_i v)
\]  

(9.7)

\[
P(v_j = 1|h) = \text{sigm}(b_j + W'_j h)
\]  

(9.8)

The free energy of an RBM with binary units further simplifies to:

\[
\mathcal{F}(v) = -b'v - \sum_i \log(1 + e^{(c_i + W_i v)}).
\]  

(9.9)
Update Equations with Binary Units

Combining Eqs. (9.5) with (9.9), we obtain the following log-likelihood gradients for an RBM with binary units:

\[
\begin{align*}
\frac{- \partial \log p(v)}{\partial W_{ij}} &= E_v[p(h_i | v) \cdot v_j] - v_j^{(i)} \cdot \text{sigmoid}(W_i \cdot v^{(i)} + c_i) \\
\frac{- \partial \log p(v)}{\partial c_i} &= E_v[p(h_i | v)] - \text{sigmoid}(W_i \cdot v^{(i)}) \\
\frac{- \partial \log p(v)}{\partial b_j} &= E_v[p(v_j | h)] - v_j^{(i)}
\end{align*}
\]  

(9.10)

For a more detailed derivation of these equations, we refer the reader to the following page, or to section 5 of Learning Deep Architectures for AI. We will however not use these formulas, but rather get the gradient using Theano T.grad from equation (9.4).

9.3 Sampling in an RBM

Samples of \( p(x) \) can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator.

Gibbs sampling of the joint of \( N \) random variables \( S = (S_1, ..., S_N) \) is done through a sequence of \( N \) sampling sub-steps of the form \( S_i \sim p(S_i | S_{-i}) \) where \( S_{-i} \) contains the \( N - 1 \) other random variables in \( S \) excluding \( S_i \).

For RBMs, \( S \) consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visible. A step in the Markov chain is thus taken as follows:

\[
\begin{align*}
h^{(n+1)} &\sim \text{sigmoid}(W'v^{(n)} + c) \\
v^{(n+1)} &\sim \text{sigmoid}(Wh^{(n+1)} + b),
\end{align*}
\]

where \( h^{(n)} \) refers to the set of all hidden units at the \( n \)-th step of the Markov chain. What it means is that, for example, \( h_i^{(n+1)} \) is randomly chosen to be 1 (versus 0) with probability \( \text{sigmoid}(W'_i v^{(n)} + c_i) \), and similarly, \( v_j^{(n+1)} \) is randomly chosen to be 1 (versus 0) with probability \( \text{sigmoid}(W_j h^{(n+1)} + b_j) \).

This can be illustrated graphically:
As \( t \to \infty \), samples \( (v(t), h(t)) \) are guaranteed to be accurate samples of \( p(v, h) \).

In theory, each parameter update in the learning process would require running one such chain to convergence. It is needless to say that doing so would be prohibitively expensive. As such, several algorithms have been devised for RBMs, in order to efficiently sample from \( p(v, h) \) during the learning process.

### 9.3.1 Contrastive Divergence (CD-k)

Contrastive Divergence uses two tricks to speed up the sampling process:

- since we eventually want \( p(v) \approx p_{\text{train}}(v) \) (the true, underlying distribution of the data), we initialize the Markov chain with a training example (i.e., from a distribution that is expected to be close to \( p \), so that the chain will be already close to having converged to its final distribution \( p \)).

- CD does not wait for the chain to converge. Samples are obtained after only \( k \)-steps of Gibbs sampling. In practice, \( k = 1 \) has been shown to work surprisingly well.

### 9.3.2 Persistent CD

Persistent CD [Tieleman08] uses another approximation for sampling from \( p(v, h) \). It relies on a single Markov chain, which has a persistent state (i.e., not restarting a chain for each observed example). For each parameter update, we extract new samples by simply running the chain for \( k \)-steps. The state of the chain is then preserved for subsequent updates.

The general intuition is that if parameter updates are small enough compared to the mixing rate of the chain, the Markov chain should be able to “catch up” to changes in the model.

### 9.4 Implementation

We construct an `RBM` class. The parameters of the network can either be initialized by the constructor or can be passed as arguments. This option is useful when an RBM is used as the building block of a deep network, in which case the weight matrix and the hidden layer bias is shared with the corresponding sigmoidal layer of an MLP network.

```python
class RBM(object):
    """Restricted Boltzmann Machine (RBM) """
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
        """RBMs constructor. Defines the parameters of the model along with
```

9.4. Implementation
basic operations for inferring hidden from visible (and vice-versa), as well as for performing CD updates.

:param input: None for standalone RBMs or symbolic variable if RBM is part of a larger graph.

:param n_visible: number of visible units

:param n_hidden: number of hidden units

:param W: None for standalone RBMs or symbolic variable pointing to a shared weight matrix in case RBM is part of a DBN network; in a DBN, the weights are shared between RBMs and layers of a MLP

:param hbias: None for standalone RBMs or symbolic variable pointing to a shared hidden units bias vector in case RBM is part of a different network

:param vbias: None for standalone RBMs or a symbolic variable pointing to a shared visible units bias

""

self.n_visible = n_visible
self.n_hidden = n_hidden

if numpy_rng is None:
    # create a number generator
    numpy_rng = numpy.random.RandomState(1234)

if theano_rng is None:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

if W is None:
    # W is initialized with 'initial_W' which is uniformly
    # sampled from -4*sqrt(6./(n_visible+n_hidden)) and
    # 4*sqrt(6./(n_hidden+n_visible)) the output of uniform if
    # converted using asarray to dtype theano.config.floatX so
    # that the code is runable on GPU
    initial_W = numpy.asarray(
        numpy_rng.uniform(
            low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            size=(n_visible, n_hidden)
        ),
        dtype=theano.config.floatX
    )
    W = theano.shared(value=initial_W, name='W', borrow=True)

if hbias is None:
    # create shared variable for hidden units bias
    hbias = theano.shared(
        value=numpy.zeros(
Next step is to define functions which construct the symbolic graph associated with Eqs. (9.7) - (9.8). The code is as follows:

```python
def propup(self, vis):
    '''This function propagates the visible units activation upwards to the hidden units

    Note that we return also the pre-sigmoid activation of the layer. As it will turn out later, due to how Theano deals with optimizations, this symbolic variable will be needed to write down a more stable computational graph (see details in the reconstruction cost function)
    '''
    pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
    return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]

def sample_h_given_v(self, v0_sample):
    ''' This function infers state of hidden units given visible units '''
    # compute the activation of the hidden units given a sample of
```
We can then use these functions to define the symbolic graph for a Gibbs sampling step. We define two functions:

- **gibbs_vhv** which performs a step of Gibbs sampling starting from the visible units. As we shall see, this will be useful for sampling from the RBM.
- **gibbs_hvh** which performs a step of Gibbs sampling starting from the hidden units. This function will be useful for performing CD and PCD updates.

The code is as follows:

```python
def gibbs_hvh(self, h0_sample):
    ''' This function implements one step of Gibbs sampling, starting from the hidden state'''
    ...
```
Note that we also return the pre-sigmoid activation. To understand why this is so you need to understand a bit about how Theano works. Whenever you compile a Theano function, the computational graph that you pass as input gets optimized for speed and stability. This is done by changing several parts of the subgraphs with others. One such optimization expresses terms of the form \( \log(\text{sigmoid}(x)) \) in terms of softplus. We need this optimization for the cross-entropy since sigmoid of numbers larger than 30. (or even less then that) turn to 1. and numbers smaller than -30. turn to 0 which in terms will force theano to compute \( \log(0) \) and therefore we will get either -inf or NaN as cost. If the value is expressed in terms of softplus we do not get this undesirable behaviour. This optimization usually works fine, but here we have a special case. The sigmoid is applied inside the scan op, while the log is outside. Therefore Theano will only see \( \log(\text{scan}(..)) \) instead of \( \log(\text{sigmoid}(..)) \) and will not apply the wanted optimization. We can not go and replace the sigmoid in scan with something else also, because this only needs to be done on the last step. Therefore the easiest and more efficient way is to get also the pre-sigmoid activation as an output of scan, and apply both the log and sigmoid outside scan such that Theano can catch and optimize the expression.

The class also has a function that computes the free energy of the model, needed for computing the gradient of the parameters (see Eq. (9.4)). Note that we also return the pre-sigmoid

We then add a \texttt{get\_cost\_updates} method, whose purpose is to generate the symbolic gradients for CD-k and PCD-k updates.
Returns a proxy for the cost and the updates dictionary. The dictionary contains the update rules for weights and biases but also an update of the shared variable used to store the persistent chain, if one is used.

```python
# compute positive phase
pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)

# decide how to initialize persistent chain:
# for CD, we use the newly generate hidden sample
# for PCD, we initialize from the old state of the chain
if persistent is None:
    chain_start = ph_sample
else:
    chain_start = persistent
```

Note that `get_cost_updates` takes as argument a variable called `persistent`. This allows us to use the same code to implement both CD and PCD. To use PCD, `persistent` should refer to a shared variable which contains the state of the Gibbs chain from the previous iteration.

If `persistent` is `None`, we initialize the Gibbs chain with the hidden sample generated during the positive phase, therefore implementing CD. Once we have established the starting point of the chain, we can then compute the sample at the end of the Gibbs chain, sample that we need for getting the gradient (see Eq. (9.4)). To do so, we will use the `scan` op provided by Theano, therefore we urge the reader to look it up by following this link.

```python
# perform actual negative phase
# in order to implement CD-k/PCD-k we need to scan over the
# function that implements one gibbs step k times.
# Read Theano tutorial on scan for more information :
# http://deeplearning.net/software/theano/library/scan.html
# the scan will return the entire Gibbs chain
([pre_sigmoid_nvs,
  nv_means,
  nv_samples,
  pre_sigmoid_nhs,
  nh_means,
  nh_samples
],
updates) = theano.scan(
    self.gibbs_hvh,
    # the None are place holders, saying that
    # chain_start is the initial state corresponding to the
    # 6th output
    outputs_info=[None, None, None, None, None, chain_start],
    n_steps=k,
    name="gibbs_hvh"
)```
Once we have generated the chain we take the sample at the end of the chain to get the free energy of the negative phase. Note that the `chain_end` is a symbolical Theano variable expressed in terms of the model parameters, and if we would apply `T.grad` naively, the function will try to go through the Gibbs chain to get the gradients. This is not what we want (it will mess up our gradients) and therefore we need to indicate to `T.grad` that `chain_end` is a constant. We do this by using the argument `consider_constant` of `T.grad`.

```python
# determine gradients on RBM parameters
# note that we only need the sample at the end of the chain
chain_end = nv_samples[-1]

cost = T.mean(self.free_energy(self.input)) - T.mean(
    self.free_energy(chain_end))
# We must not compute the gradient through the gibbs sampling

gparams = T.grad(cost, self.params, consider_constant=[chain_end])
```

Finally, we add to the updates dictionary returned by scan (which contains updates rules for random states of `theano_rng`) to contain the parameter updates. In the case of PCD, these should also update the shared variable containing the state of the Gibbs chain.

```python
# constructs the update dictionary
for gparam, param in zip(gparams, self.params):
    # make sure that the learning rate is of the right dtype
    updates[param] = param - gparam * T.cast(lr, dtype=theano.config.floatX)

if persistent:
    # Note that this works only if persistent is a shared variable
    updates[persistent] = nh_samples[-1]
# pseudo-likelihood is a better proxy for PCD
monitoring_cost = self.get_pseudo_likelihood_cost(updates)
else:
    # reconstruction cross-entropy is a better proxy for CD
    monitoring_cost = self.get_reconstruction_cost(updates, pre_sigmoid_nvs[-1])

return monitoring_cost, updates
```

### 9.4.1 Tracking Progress

RBM s are particularly tricky to train. Because of the partition function $Z$ of Eq. (9.1), we cannot estimate the log-likelihood $\log(P(x))$ during training. We therefore have no direct useful metric for choosing the optimal hyperparameters.

Several options are available to the user.

**Inspection of Negative Samples**

---

9.4. Implementation
Negative samples obtained during training can be visualized. As training progresses, we know that the model defined by the RBM becomes closer to the true underlying distribution, $p_{\text{train}}(x)$. Negative samples should thus look like samples from the training set. Obviously bad hyperparameters can be discarded in this fashion.

**Visual Inspection of Filters**

The filters learnt by the model can be visualized. This amounts to plotting the weights of each unit as a gray-scale image (after reshaping to a square matrix). Filters should pick out strong features in the data. While it is not clear for an arbitrary dataset, what these features should look like, training on MNIST usually results in filters which act as stroke detectors, while training on natural images lead to Gabor like filters if trained in conjunction with a sparsity criteria.

**Proxies to Likelihood**

Other, more tractable functions can be used as a proxy to the likelihood. When training an RBM with PCD, one can use pseudo-likelihood as the proxy. Pseudo-likelihood (PL) is much less expensive to compute, as it assumes that all bits are independent. Therefore,

$$PL(x) = \prod_i P(x_i|x_{-i})$$

$$\log PL(x) = \sum_i \log P(x_i|x_{-i})$$

Here $x_{-i}$ denotes the set of all bits of $x$ except bit $i$. The log-PL is therefore the sum of the log-probabilities of each bit $x_i$, conditioned on the state of all other bits. For MNIST, this would involve summing over the 784 input dimensions, which remains rather expensive. For this reason, we use the following stochastic approximation to log-PL:

$$g = N \cdot \log P(x_i|x_{-i}), \text{ where } i \sim U(0, N), \text{ and}$$

$$E[g] = \log PL(x)$$

where the expectation is taken over the uniform random choice of index $i$, and $N$ is the number of visible units. In order to work with binary units, we further introduce the notation $\tilde{x}_i$ to refer to $x$ with bit-$i$ being flipped (1->0, 0->1). The log-PL for an RBM with binary units is then written as:

$$\log PL(x) \approx N \cdot \log \frac{e^{-FE(x)}}{e^{-FE(x)} + e^{-FE(\tilde{x}_i)}}$$

$$\approx N \cdot \log[\text{sigmoid}(FE(\tilde{x}_i) - FE(x))]$$

We therefore return this cost as well as the RBM updates in the `get_cost_updates` function of the RBM class. Notice that we modify the updates dictionary to increment the index of bit $i$. This will result in bit $i$ cycling over all possible values \{0, 1, ..., $N$\}, from one update to another.

Note that for CD training the cross-entropy cost between the input and the reconstruction (the same as the one used for the de-noising autoencoder) is more reliable then the pseudo-loglikelihood. Here is the code we use to compute the pseudo-likelihood:

```python
def get_pseudo_likelihood_cost(self, updates):
    """Stochastic approximation to the pseudo-likelihood""

    # index of bit i in expression p(x_i | x_{\setminus i})
```
bit_i_idx = theano.shared(value=0, name='bit_i_idx')

# binarize the input image by rounding to nearest integer
xi = T.round(self.input)

# calculate free energy for the given bit configuration
fe_xi = self.free_energy(xi)

# flip bit x_i of matrix xi and preserve all other bits x_{\text{i} \neq i}
# Equivalent to xi[:,bit_i_idx] = 1-xi[:, bit_i_idx], but assigns
# the result to xi_flip, instead of working in place on xi.
xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])

# calculate free energy with bit flipped
fe_xi_flip = self.free_energy(xi_flip)

# equivalent to e^{-FE(x_i)} / (e^{-FE(x_i)} + e^{-FE(x_{\text{i} \neq i})})
# increment bit_i_idx % number as part of updates
cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip - fe_xi)))
updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

return cost

9.4.2 Main Loop

We now have all the necessary ingredients to start training our network.

Before going over the training loop however, the reader should familiarize himself with the function tile_raster_images (see Plotting Samples and Filters). Since RBMs are generative models, we are interested in sampling from them and plotting/visualizing these samples. We also want to visualize the filters (weights) learnt by the RBM, to gain insights into what the RBM is actually doing. Bear in mind however, that this does not provide the entire story, since we neglect the biases and plot the weights up to a multiplicative constant (weights are converted to values between 0 and 1).

Having these utility functions, we can start training the RBM and plot/save the filters after each training epoch. We train the RBM using PCD, as it has been shown to lead to a better generative model ([Tieleman08]).

# it is ok for a theano function to have no output
# the purpose of train_rbm is solely to update the RBM parameters
train_rbm = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size]
    },
    name='train_rbm'
)
Once the RBM is trained, we can then use the `gibbs_vhv` function to implement the Gibbs chain required for sampling. We initialize the Gibbs chain starting from test examples (although we could as well pick it from the training set) in order to speed up convergence and avoid problems with random initialization. We again use Theano's `scan` op to do 1000 steps before each plotting.
Next we create the 20 persistent chains in parallel to get our samples. To do so, we compile a theano function which performs one Gibbs step and updates the state of the persistent chain with the new visible sample. We apply this function iteratively for a large number of steps, plotting the samples at every 1000 steps.

```python
plot_every = 1000
# define one step of Gibbs sampling (mf = mean-field) define a # function that does `plot_every` steps before returning the # sample for plotting
(plot
    [ presig_hids,
      hid_mfs,
      hid_samples,
      presig_vis,
      vis_mfs,
      vis_samples
    ],
    updates
) = theano.scan(
    rbm.gibbs_vhv,
    outputs_info=[None, None, None, None, None, persistent_vis_chain],
    n_steps=plot_every,
    name="gibbs_vhv"
)

# add to updates the shared variable that takes care of our persistent # chain :
updates.update({persistent_vis_chain: vis_samples[-1]})
# construct the function that implements our persistent chain.
# we generate the "mean field" activations for plotting and the actual # samples for reinitializing the state of our persistent chain
sample_fn = theano.function(
    [],
    [ vis_mfs[-1],
      vis_samples[-1]
    ],
    updates=updates,
    name='sample_fn'
)

# create a space to store the image for plotting ( we need to leave # room for the tile_spacing as well)
image_data = numpy.zeros(
    (29 * n_samples + 1, 29 * n_chains - 1),
    dtype='uint8'
)
for idx in range(n_samples):
    # generate `plot_every` intermediate samples that we discard,
    # because successive samples in the chain are too correlated
    vis_mf, vis_sample = sample_fn()
9.5 Results

We ran the code with PCD-15, learning rate of 0.1 and a batch size of 20, for 15 epochs. Training the model takes 122.466 minutes on a Intel Xeon E5430 @ 2.66GHz CPU, with a single-threaded GotoBLAS.

The output was the following:

```
... loading data
Training epoch 0, cost is -90.6507246003
Training epoch 1, cost is -81.235857373
Training epoch 2, cost is -74.9120966945
Training epoch 3, cost is -73.0213216101
Training epoch 4, cost is -68.4098570497
Training epoch 5, cost is -63.2693021647
Training epoch 6, cost is -65.99578971
Training epoch 7, cost is -68.1236650015
Training epoch 8, cost is -68.3207365087
Training epoch 9, cost is -64.2949797113
Training epoch 10, cost is -61.5194867893
Training epoch 11, cost is -61.6539369402
Training epoch 12, cost is -63.5465278086
Training epoch 13, cost is -63.3787093527
Training epoch 14, cost is -62.755739271
Training took 122.466000 minutes
... plotting sample 0
... plotting sample 1
... plotting sample 2
... plotting sample 3
... plotting sample 4
... plotting sample 5
... plotting sample 6
... plotting sample 7
... plotting sample 8
... plotting sample 9
```

The pictures below show the filters after 15 epochs:

Here are the samples generated by the RBM after training. Each row represents a mini-batch of negative particles (samples from independent Gibbs chains). 1000 steps of Gibbs sampling were taken between each
Fig. 9.1: Filters obtained after 15 epochs.

of those rows.
Deep Belief Networks

Note: This section assumes the reader has already read through *Classifying MNIST digits using Logistic Regression* and *Multilayer Perceptron and Restricted Boltzmann Machines (RBM)*. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX. If you intend to run the code on GPU also read GPU.

Note: The code for this section is available for download here.

### 10.1 Deep Belief Networks

[Hinton06] showed that RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks (DBN). DBNs are graphical models which learn to extract a deep hierarchical representation of the training data. They model the joint distribution between observed vector $x$ and the $\ell$ hidden layers $h^k$ as follows:

$$P(x, h^1, \ldots, h^\ell) = \left( \prod_{k=0}^{\ell-2} P(h^k|h^{k+1}) \right) P(h^{\ell-1}, h^\ell)$$  \hspace{1cm} (10.1)

where $x = h^0$, $P(h^{k-1}|h^k)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level $k$, and $P(h^{\ell-1}, h^\ell)$ is the visible-hidden joint distribution in the top-level RBM. This is illustrated in the figure below.

The principle of greedy layer-wise unsupervised training can be applied to DBNs with RBMs as the building blocks for each layer [Hinton06], [Bengio07]. The process is as follows:

1. Train the first layer as an RBM that models the raw input $x = h^{(0)}$ as its visible layer.
2. Use that first layer to obtain a representation of the input that will be used as data for the second layer. Two common solutions exist. This representation can be chosen as being the mean activations \( p(h^{(1)} = 1 | h^{(0)}) \) or samples of \( p(h^{(1)} | h^{(0)}) \).

3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM).

4. Iterate (2 and 3) for the desired number of layers, each time propagating upward either samples or mean values.

5. Fine-tune all the parameters of this deep architecture with respect to a proxy for the DBN log-likelihood, or with respect to a supervised training criterion (after adding extra learning machinery to convert the learned representation into supervised predictions, e.g. a linear classifier).

In this tutorial, we focus on fine-tuning via supervised gradient descent. Specifically, we use a logistic regression classifier to classify the input \( x \) based on the output of the last hidden layer \( h^{(l)} \) of the DBN. Fine-tuning is then performed via supervised gradient descent of the negative log-likelihood cost function. Since the supervised gradient is only non-null for the weights and hidden layer biases of each layer (i.e. null for the visible biases of each RBM), this procedure is equivalent to initializing the parameters of a deep MLP with the weights and hidden layer biases obtained with the unsupervised training strategy.

### 10.2 Justifying Greedy-Layer Wise Pre-Training

Why does such an algorithm work? Taking as example a 2-layer DBN with hidden layers \( h^{(1)} \) and \( h^{(2)} \) (with respective weight parameters \( W^{(1)} \) and \( W^{(2)} \)), [Hinton06] established (see also Bengio09] for a detailed derivation) that \( \log p(x) \) can be rewritten as,

\[
\log p(x) = KL(Q(h^{(1)}|x)||p(h^{(1)}|x)) + H_{Q(h^{(1)}|x)} + \sum_h Q(h^{(1)}|x)(\log p(h^{(1)}) + \log p(x|h^{(1)})). \tag{10.2}
\]

\( KL(Q(h^{(1)}|x)||p(h^{(1)}|x)) \) represents the KL divergence between the posterior \( Q(h^{(1)}|x) \) of the first RBM if it were standalone, and the probability \( p(h^{(1)}|x) \) for the same layer but defined by the entire DBN (i.e.
taking into account the prior $p(h^{(1)}, h^{(2)})$ defined by the top-level RBM. $H_{Q(h^{(1)}|x)}$ is the entropy of the distribution $Q(h^{(1)}|x)$.

It can be shown that if we initialize both hidden layers such that $W^{(2)} = W^{(1)^T}$, $Q(h^{(1)}|x) = p(h^{(1)}|x)$ and the KL divergence term is null. If we learn the first level RBM and then keep its parameters $W^{(1)}$ fixed, optimizing Eq. (10.2) with respect to $W^{(2)}$ can thus only increase the likelihood $p(x)$.

Also, notice that if we isolate the terms which depend only on $W^{(2)}$, we get:

$$
\sum_h Q(h^{(1)}|x)p(h^{(1)})
$$

Optimizing this with respect to $W^{(2)}$ amounts to training a second-stage RBM, using the output of $Q(h^{(1)}|x)$ as the training distribution, when $x$ is sampled from the training distribution for the first RBM.

### 10.3 Implementation

To implement DBNs in Theano, we will use the class defined in the *Restricted Boltzmann Machines (RBM)* tutorial. One can also observe that the code for the DBN is very similar with the one for SdA, because both involve the principle of unsupervised layer-wise pre-training followed by supervised fine-tuning as a deep MLP. The main difference is that we use the RBM class instead of the dA class.

We start off by defining the DBN class which will store the layers of the MLP, along with their associated RBMs. Since we take the viewpoint of using the RBMs to initialize an MLP, the code will reflect this by separating as much as possible the RBMs used to initialize the network and the MLP used for classification.

```python
class DBN(object):
    ""
    Deep Belief Network
    A deep belief network is obtained by stacking several RBMs on top of each other. The hidden layer of the RBM at layer 'i' becomes the input of the RBM at layer 'i+1'. The first layer RBM gets as input the input of the network, and the hidden layer of the last RBM represents the output. When used for classification, the DBN is treated as a MLP, by adding a logistic regression layer on top.
    ""

    def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
                 hidden_layers_sizes=[500, 500], n_outs=10):
        ""
        This class is made to support a variable number of layers.
        :type numpy_rng: numpy.random.RandomState
        :param numpy_rng: numpy random number generator used to draw initial weights
        :type theano_rng: theano.tensor.shared_randomstreams.RandomeStreams
        :param theano_rng: Theano random generator; if None is given one is generated based on a seed drawn from `rng`
        :type n_ins: int
        :param n_ins: dimension of the input to the DBN
        ""
```
self.sigmoid_layers will store the feed-forward graphs which together form the MLP, while `self.rbm_layers` will store the RBMs used to pretrain each layer of the MLP.

Next step, we construct `n_layers` sigmoid layers (we use the `HiddenLayer` class introduced in *Multilayer Perceptron*, with the only modification that we replaced the non-linearity from \( \tanh \) to the logistic function \( s(x) = \frac{1}{1+e^{-x}} \)) and `n_layers` RBMs, where `n_layers` is the depth of our model. We link the sigmoid layers such that they form an MLP, and construct each RBM such that they share the weight matrix and the hidden bias with its corresponding sigmoid layer.

```python
for i in range(self.n_layers):
    # construct the sigmoidal layer
    # the size of the input is either the number of hidden
    # units of the layer below or the input size if we are on
    # the first layer
    if i == 0:
        input_size = n_ins
    else:
        input_size = hidden_layers_sizes[i - 1]

    # the input to this layer is either the activation of the
    # hidden layer below or the input of the DBN if you are on
    # the first layer
    if i == 0:
        layer_input = self.x
```
else:
    layer_input = self.sigmoid_layers[-1].output

sigmoid_layer = HiddenLayer(rng=numpy_rng,
    input=layer_input,
    n_in=input_size,
    n_out=hidden_layers_sizes[i],
    activation=T.nnet.sigmoid)

# add the layer to our list of layers
self.sigmoid_layers.append(sigmoid_layer)

# its arguably a philosophical question... but we are
# going to only declare that the parameters of the
# sigmoid_layers are parameters of the DBN. The visible
# biases in the RBM are parameters of those RBMs, but not
# of the DBN.
self.params.extend(sigmoid_layer.params)

# Construct an RBM that shared weights with this layer
rbm_layer = RBM(numpy_rng=numpy_rng,
    theano_rng=theano_rng,
    input=layer_input,
    n_visible=input_size,
    n_hidden=hidden_layers_sizes[i],
    W=sigmoid_layer.W,
    hbias=sigmoid_layer.b)
self.rbm_layers.append(rbm_layer)

All that is left is to stack one last logistic regression layer in order to form an MLP. We will use the
LogisticRegression class introduced in Classifying MNIST digits using Logistic Regression.

self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs)
self.params.extend(self.logLayer.params)

# compute the cost for second phase of training, defined as the
# negative log likelihood of the logistic regression (output) layer
self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)

# compute the gradients with respect to the model parameters
# symbolic variable that points to the number of errors made on the
# minibatch given by self.x and self.y
self.errors = self.logLayer.errors(self.y)

The class also provides a method which generates training functions for each of the RBMs. They are returned
as a list, where element $i$ is a function which implements one step of training for the RBM at layer $i$. 

10.3. Implementation
def pretraining_functions(self, train_set_x, batch_size, k):
    '''Generates a list of functions, for performing one step of
gradient descent at a given layer. The function will require
as input the minibatch index, and to train an RBM you just
need to iterate, calling the corresponding function on all
minibatch indexes.

:type train_set_x: theano.tensor.TensorType
:param train_set_x: Shared var. that contains all datapoints used
    for training the RBM
:type batch_size: int
:param batch_size: size of a [mini]batch
:param k: number of Gibbs steps to do in CD-k / PCD-k
    '''

    index = T.lscalar('index')  # index to a [mini]batch
    index = T.lscalar('index')  # index to a minibatch

    # index to a [mini]batch
    index = T.lscalar('index')  # index to a minibatch

    In order to be able to change the learning rate during training, we associate a Theano variable to it that has
a default value.

    learning_rate = T.scalar('lr')  # learning rate to use

    # begining of a batch, given `index`
    batch_begin = index * batch_size
    # ending of a batch given `index`
    batch_end = batch_begin + batch_size

    pretrain_fns = []
    for rbm in self.rbm_layers:
        # get the cost and the updates list
        # using CD-k here (persisent=None) for training each RBM.
        # TODO: change cost function to reconstruction error
        cost, updates = rbm.get_cost_updates(learning_rate,
                                              persistent=None, k=k)

        # compile the theano function
        fn = theano.function(
            inputs=[index, theano.In(learning_rate, value=0.1)],
            outputs=cost,
            updates=updates,
            givens={
                self.x: train_set_x[batch_begin:batch_end]
            }
        )

        # append `fn` to the list of functions
        pretrain_fns.append(fn)

    return pretrain_fns
Now any function `pretrain_fns[i]` takes as arguments `index` and optionally `lr` - the learning rate. Note that the names of the parameters are the names given to the Theano variables (e.g. `lr`) when they are constructed and not the name of the python variables (e.g. `learning_rate`). Keep this in mind when working with Theano. Optionally, if you provide `k` (the number of Gibbs steps to perform in CD or PCD) this will also become an argument of your function.

In the same fashion, the DBN class includes a method for building the functions required for finetuning (a `train_model`, a `validate_model` and a `test_model` function).

```python
def build_finetune_functions(self, datasets, batch_size, learning_rate):
    '''Generates a function `train` that implements one step of finetuning, a function `validate` that computes the error on a batch from the validation set, and a function `test` that computes the error on a batch from the testing set
    :
    :type datasets: list of pairs of theano.tensor.TensorType
    :param datasets: It is a list that contain all the datasets;
        the has to contain three pairs, `train`, `valid`, `test` in this order, where each pair
        is formed of two Theano variables, one for the datapoints, the other for the labels
    :
    :type batch_size: int
    :param batch_size: size of a minibatch
    :type learning_rate: float
    :param learning_rate: learning rate used during finetune stage
    '''
    (train_set_x, train_set_y) = datasets[0]
    (valid_set_x, valid_set_y) = datasets[1]
    (test_set_x, test_set_y) = datasets[2]

    # compute number of minibatches for training, validation and testing
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
    n_valid_batches //= batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0]
    n_test_batches //= batch_size
    index = T.lscalar('index')  # index to a [mini]batch

    # compute the gradients with respect to the model parameters
    gparams = T.grad(self.finetune_cost, self.params)

    # compute list of fine-tuning updates
    updates = []
    for param, gparam in zip(self.params, gparams):
        updates.append((param, param - gparam * learning_rate))

    train_fn = theano.function(
        inputs=[index],
        outputs=self.finetune_cost,
        updates=updates,
        givens={
```
self.x: train_set_x[
    index * batch_size: (index + 1) * batch_size
],
self.y: train_set_y[
    index * batch_size: (index + 1) * batch_size
]
}
)

test_score_i = theano.function(
    [index],
    self.errors,
    givens={
    self.x: test_set_x[
        index * batch_size: (index + 1) * batch_size
    ],
    self.y: test_set_y[
        index * batch_size: (index + 1) * batch_size
    ]
}
)

valid_score_i = theano.function(
    [index],
    self.errors,
    givens={
    self.x: valid_set_x[
        index * batch_size: (index + 1) * batch_size
    ],
    self.y: valid_set_y[
        index * batch_size: (index + 1) * batch_size
    ]
}
)

# Create a function that scans the entire validation set
def valid_score():
    return [valid_score_i(i) for i in range(n_valid_batches)]

# Create a function that scans the entire test set
def test_score():
    return [test_score_i(i) for i in range(n_test_batches)]

return train_fn, valid_score, test_score

Note that the returned valid_score and test_score are not Theano functions, but rather Python functions. These loop over the entire validation set and the entire test set to produce a list of the losses obtained over these sets.
10.4 Putting it all together

The few lines of code below constructs the deep belief network:

```
numpy_rng = numpy.random.RandomState(123)
print('... building the model')
# construct the Deep Belief Network
dbn = DBN(numpy_rng=numpy_rng, n_ins=28 * 28,
             hidden_layers_sizes=[1000, 1000, 1000],
             n_outs=10)
```

There are two stages in training this network: (1) a layer-wise pre-training and (2) a fine-tuning stage.

For the pre-training stage, we loop over all the layers of the network. For each layer, we use the compiled theano function which determines the input to the i-th level RBM and performs one step of CD-k within this RBM. This function is applied to the training set for a fixed number of epochs given by `pretraining_epochs`.

```
# PRETRAINING THE MODEL #
print('... getting the pretraining functions')
pretraining_fns = dbn.pretraining_functions(train_set_x=train_set_x,
                                          batch_size=batch_size,
                                          k=k)
print('... pre-training the model')
start_time = timeit.default_timer()
# Pre-train layer-wise
for i in range(dbn.n_layers):
    # go through pretraining epochs
    for epoch in range(pretraining_epochs):
        # go through the training set
        c = []
        for batch_index in range(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
                             lr=pretrain_lr))
        print('Pre-training layer %i, epoch %d, cost %f (i, epoch), end=' % (i, epoch, end=
              → '))
        print(numpy.mean(c, dtype='float64'))
end_time = timeit.default_timer()  
```

The fine-tuning loop is very similar to the one in the `Multilayer Perceptron` tutorial, the only difference being that we now use the functions given by `build_finetune_functions`.

10.5 Running the Code

The user can run the code by calling:
With the default parameters, the code runs for 100 pre-training epochs with mini-batches of size 10. This corresponds to performing 500,000 unsupervised parameter updates. We use an unsupervised learning rate of 0.01, with a supervised learning rate of 0.1. The DBN itself consists of three hidden layers with 1000 units per layer. With early-stopping, this configuration achieved a minimal validation error of 1.27 with corresponding test error of 1.34 after 46 supervised epochs.

On an Intel(R) Xeon(R) CPU X5560 running at 2.80GHz, using a multi-threaded MKL library (running on 4 cores), pretraining took 615 minutes with an average of 2.05 mins/(layer * epoch). Fine-tuning took only 101 minutes or approximately 2.20 mins/epoch.

Hyper-parameters were selected by optimizing on the validation error. We tested unsupervised learning rates in \( \{10^{-1}, ..., 10^{-5}\}\) and supervised learning rates in \( \{10^{-1}, ..., 10^{-4}\}\). We did not use any form of regularization besides early-stopping, nor did we optimize over the number of pretraining updates.

### 10.6 Tips and Tricks

One way to improve the running time of your code (given that you have sufficient memory available), is to compute the representation of the entire dataset at layer \( i \) in a single pass, once the weights of the \( i - 1 \)-th layers have been fixed. Namely, start by training your first layer RBM. Once it is trained, you can compute the hidden units values for every example in the dataset and store this as a new dataset which is used to train the 2nd layer RBM. Once you trained the RBM for layer 2, you compute, in a similar fashion, the dataset for layer 3 and so on. This avoids calculating the intermediate (hidden layer) representations, \( \text{pretraining\_epochs} \) times at the expense of increased memory usage.
CHAPTER 11

Hybrid Monte-Carlo Sampling

Note: This is an advanced tutorial, which shows how one can implemented Hybrid Monte-Carlo (HMC) sampling using Theano. We assume the reader is already familiar with Theano and energy-based models such as the RBM.

Note: The code for this section is available for download here.

11.1 Theory

Maximum likelihood learning of energy-based models requires a robust algorithm to sample negative phase particles (see Eq.(4) of the Restricted Boltzmann Machines (RBM) tutorial). When training RBMs with CD or PCD, this is typically done with block Gibbs sampling, where the conditional distributions $p(h|v)$ and $p(v|h)$ are used as the transition operators of the Markov chain.

In certain cases however, these conditional distributions might be difficult to sample from (i.e. requiring expensive matrix inversions, as in the case of the “mean-covariance RBM”). Also, even if Gibbs sampling can be done efficiently, it nevertheless operates via a random walk which might not be statistically efficient for some distributions. In this context, and when sampling from continuous variables, Hybrid Monte Carlo (HMC) can prove to be a powerful tool [Duane87]. It avoids random walk behavior by simulating a physical system governed by Hamiltonian dynamics, potentially avoiding tricky conditional distributions in the process.

In HMC, model samples are obtained by simulating a physical system, where particles move about a high-dimensional landscape, subject to potential and kinetic energies. Adapting the notation from [Neal93], particles are characterized by a position vector or state $s \in \mathcal{R}^D$ and velocity vector $\phi \in \mathcal{R}^D$. The combined
state of a particle is denoted as \( \chi = (s, \phi) \). The Hamiltonian is then defined as the sum of potential energy \( E(s) \) (same energy function defined by energy-based models) and kinetic energy \( K(\phi) \), as follows:

\[
\mathcal{H}(s, \phi) = E(s) + K(\phi) = E(s) + \frac{1}{2} \sum_i \phi_i^2
\]

Instead of sampling \( p(s) \) directly, HMC operates by sampling from the canonical distribution \( p(s, \phi) = \frac{1}{Z} \exp(-\mathcal{H}(s, \phi)) = p(s)p(\phi) \). Because the two variables are independent, marginalizing over \( \phi \) is trivial and recovers the original distribution of interest.

### Hamiltonian Dynamics

State \( s \) and velocity \( \phi \) are modified such that \( \mathcal{H}(s, \phi) \) remains constant throughout the simulation. The differential equations are given by:

\[
\begin{align*}
\frac{ds_i}{dt} &= \frac{\partial \mathcal{H}}{\partial \phi_i} = \phi_i \\
\frac{d\phi_i}{dt} &= -\frac{\partial \mathcal{H}}{\partial s_i} = -\frac{\partial E}{\partial s_i}
\end{align*}
\]

As shown in [Neal93], the above transformation preserves volume and is reversible. The above dynamics can thus be used as transition operators of a Markov chain and will leave \( p(s, \phi) \) invariant. That chain by itself is not ergodic however, since simulating the dynamics maintains a fixed Hamiltonian \( \mathcal{H}(s, \phi) \). HMC thus alternates hamiltonian dynamic steps, with Gibbs sampling of the velocity. Because \( p(s) \) and \( p(\phi) \) are independent, sampling \( \phi_{\text{new}} \sim p(\phi|s) \) is trivial since \( p(\phi|s) = p(\phi) \), where \( p(\phi) \) is often taken to be the uni-variate Gaussian.

### The Leap-Frog Algorithm

In practice, we cannot simulate Hamiltonian dynamics exactly because of the problem of time discretization. There are several ways one can do this. To maintain invariance of the Markov chain however, care must be taken to preserve the properties of volume conservation and time reversibility. The leap-frog algorithm maintains these properties and operates in 3 steps:

\[
\begin{align*}
\phi_i(t + \epsilon/2) &= \phi_i(t) - \frac{\epsilon}{2} \frac{\partial}{\partial s_i} E(s(t)) \\
s_i(t + \epsilon) &= s_i(t) + \epsilon \phi_i(t + \epsilon/2) \\
\phi_i(t + \epsilon) &= \phi_i(t + \epsilon/2) - \frac{\epsilon}{2} \frac{\partial}{\partial s_i} E(s(t + \epsilon))
\end{align*}
\]

We thus perform a half-step update of the velocity at time \( t + \epsilon/2 \), which is then used to compute \( s(t + \epsilon) \) and \( \phi(t + \epsilon) \).

### Accept / Reject

In practice, using finite stepsizes \( \epsilon \) will not preserve \( \mathcal{H}(s, \phi) \) exactly and will introduce bias in the simulation. Also, rounding errors due to the use of floating point numbers means that the above transformation will not be perfectly reversible.

HMC cancels these effects exactly by adding a Metropolis accept/reject stage, after \( n \) leapfrog steps. The new state \( \chi' = (s', \phi') \) is accepted with probability \( p_{\text{acc}}(\chi, \chi') \), defined as:

\[
p_{\text{acc}}(\chi, \chi') = \min \left( 1, \frac{\exp(-\mathcal{H}(s', \phi'))}{\exp(-\mathcal{H}(s, \phi))} \right)
\]
HMC Algorithm

In this tutorial, we obtain a new HMC sample as follows:

1. sample a new velocity from a univariate Gaussian distribution
2. perform $n$ leapfrog steps to obtain the new state $\chi'$
3. perform accept/reject move of $\chi'$

11.2 Implementing HMC Using Theano

In Theano, update dictionaries and shared variables provide a natural way to implement a sampling algorithm. The current state of the sampler can be represented as a Theano shared variable, with HMC updates being implemented by the updates list of a Theano function.

We breakdown the HMC algorithm into the following sub-components:

- **simulate_dynamics**: a symbolic Python function which, given an initial position and velocity, will perform $n_{steps}$ leapfrog updates and return the symbolic variables for the proposed state $\chi'$.

- **hmc_move**: a symbolic Python function which given a starting position, generates $\chi$ by randomly sampling a velocity vector. It then calls `simulate_dynamics` and determines whether the transition $\chi \rightarrow \chi'$ is to be accepted.

- **hmc_updates**: a Python function which, given the symbolic outputs of `hmc_move`, generates the list of updates for a single iteration of HMC.

- **HMC_sampler**: a Python helper class which wraps everything together.

simulate_dynamics

To perform $n$ leapfrog steps, we first need to define a function over which `Scan` can iterate over. Instead of implementing Eq. (11.2) verbatim, notice that we can obtain $s(t + n\epsilon)$ and $\phi(t + n\epsilon)$ by performing an initial half-step update for $\phi$, followed by $n$ full-step updates for $s, \phi$ and one last half-step update for $\phi$. In loop form, this gives:

\[
\begin{align*}
\phi_i(t + \epsilon/2) &= \phi_i(t) - \frac{\epsilon}{2} \frac{\partial}{\partial s_i} E(s(t)) \\
\phi_i(t + m\epsilon) &= \phi_i(t + (m - 1/2)\epsilon) - \epsilon \frac{\partial}{\partial s_i} E(s(t + (m - 1)\epsilon)) \\
s_i(t + \epsilon) &= s_i(t) + \epsilon \phi_i(t + \epsilon/2)
\end{align*}
\]

For $m \in [2, n]$, perform full updates:

\[
\begin{align*}
\phi_i(t + m\epsilon) &= \phi_i(t + (m - 3/2)\epsilon) - \epsilon \frac{\partial}{\partial s_i} E(s(t + (m - 1)\epsilon)) \\
s_i(t + m\epsilon) &= s_i(t) + \epsilon \phi_i(t + (m - 1/2)\epsilon) \\
\phi_i(t + n\epsilon) &= \phi_i(t + (n - 1/2)\epsilon) - \frac{\epsilon}{2} \frac{\partial}{\partial s_i} E(s(t + n\epsilon))
\end{align*}
\]

The inner-loop defined above is implemented by the following `leapfrog` function, with `pos, vel` and `step` replacing $s, \phi$ and $\epsilon$ respectively.
def leapfrog(pos, vel, step):
    
    """
    Inside loop of Scan. Performs one step of leapfrog update, using
    Hamiltonian dynamics.
    
    Parameters
    ----------
    pos: theano matrix
        in leapfrog update equations, represents pos(t), position at time t
    vel: theano matrix
        in leapfrog update equations, represents vel(t - stepsize/2),
        velocity at time (t - stepsize/2)
    step: theano scalar
        scalar value controlling amount by which to move
    
    Returns
    -------
    rval1: [theano matrix, theano matrix]
        Symbolic theano matrices for new position pos(t + stepsize), and
        velocity vel(t + stepsize/2)
    rval2: dictionary
        Dictionary of updates for the Scan Op
    """
    # from pos(t) and vel(t-stepsize//2), compute vel(t+stepsize//2)
    dE_dpos = TT.grad(energy_fn(pos).sum(), pos)
    new_vel = vel - step * dE_dpos
    # from vel(t+stepsize//2) compute pos(t+stepsize)
    new_pos = pos + step * new_vel
    return [new_pos, new_vel], {}

The simulate_dynamics function performs the full algorithm of Eqs. (11.3). We start with the initial half-step update of $\phi$ and full-step of $s$, and then scan over the leapfrog method $n_{steps} - 1$ times.

def simulate_dynamics(initial_pos, initial_vel, stepsize, n_steps, energy_fn):
    
    """
    Return final (position, velocity) obtained after an `n_steps` leapfrog
    updates, using Hamiltonian dynamics.
    
    Parameters
    ----------
    initial_pos: shared theano matrix
        Initial position at which to start the simulation
    initial_vel: shared theano matrix
        Initial velocity of particles
    stepsize: shared theano scalar
        Scalar value controlling amount by which to move
    energy_fn: python function
        Python function, operating on symbolic theano variables, used to
        compute the potential energy at a given position.
    
    Returns
    -------
    """
rval1: theano matrix
Final positions obtained after simulation
rval2: theano matrix
Final velocity obtained after simulation

```
def leapfrog(pos, vel, step):
    ""
    Inside loop of Scan. Performs one step of leapfrog update, using
    Hamiltonian dynamics.
    Parameters
    ----------
    pos: theano matrix
        in leapfrog update equations, represents pos(t), position at time t
    vel: theano matrix
        in leapfrog update equations, represents vel(t - stepsize/2),
        velocity at time (t - stepsize/2)
    step: theano scalar
        scalar value controlling amount by which to move
    Returns
    -------
    rval1: [theano matrix, theano matrix]
        Symbolic theano matrices for new position pos(t + stepsize), and
        velocity vel(t + stepsize/2)
    rval2: dictionary
        Dictionary of updates for the Scan Op
    ""
    # from pos(t) and vel(t-stepsize//2), compute vel(t+stepsize//2)
    dE_dpos = TT.grad(energy_fn(pos).sum(), pos)
    new_vel = vel - step * dE_dpos
    # from vel(t+stepsize//2) compute pos(t+stepsize)
    new_pos = pos + step * new_vel
    return [new_pos, new_vel], {}

    # compute velocity at time-step: t + stepsize/2
    initial_energy = energy_fn(initial_pos)
    dE_dpos = TT.grad(initial_energy.sum(), initial_pos)
    vel_half_step = initial_vel - 0.5 * stepsize * dE_dpos
    # compute position at time-step: t + stepsize
    pos_full_step = initial_pos + stepsize * vel_half_step

    # perform leapfrog updates: the scan op is used to repeatedly compute
    # vel(t + (m-1/2)*stepsize) and pos(t + m*stepsize) for m in [2,n_steps].
    (all_pos, all_vel), scan_updates = theano.scan(
        leapfrog,
        outputs_info=[
            dict(initial=pos_full_step),
            dict(initial=vel_half_step),
        ],
    )
```

11.2. Implementing HMC Using Theano
non_sequences=[stepsize],
n_steps=n_steps - 1)
final_pos = all_pos[-1]
final_vel = all_vel[-1]

# NOTE: Scan always returns an updates dictionary, in case the
# scanned function draws samples from a RandomStream. These
# updates must then be used when compiling the Theano function, to
# avoid drawing the same random numbers each time the function is
# called. In this case however, we consciously ignore
# "scan_updates" because we know it is empty.
assert not scan_updates

# The last velocity returned by scan is vel(t +
# (n_steps - 1 / 2) * stepsize) We therefore perform one more half-step
# to return vel(t + n_steps * stepsize)
energy = energy_fn(final_pos)
final_vel = final_vel - 0.5 * stepsize * TT.grad(energy.sum(), final_pos)

# return new proposal state
return final_pos, final_vel

A final half-step is performed to compute \( \phi(t + n\epsilon) \), and the final proposed state \( \chi' \) is returned.

**hmc_move**

The `hmc_move` function implements the remaining steps (steps 1 and 3) of an HMC move proposal (while wrapping the `simulate_dynamics` function). Given a matrix of initial states \( s \in \mathbb{R}^{N \times D} \) (positions) and energy function \( E(s) \) (`energy_fn`), it defines the symbolic graph for computing `n_steps` of HMC, using a given `stepsize`. The function prototype is as follows:

```python
def hmc_move(s_rng, positions, energy_fn, stepsize, n_steps):
    
    This function performs one-step of Hybrid Monte-Carlo sampling. We start by
    sampling a random velocity from a univariate Gaussian distribution,
    perform \( n\) steps' leap-frog updates using Hamiltonian dynamics and accept-reject
    using Metropolis-Hastings.

    Parameters
    ----------
    s_rng: theano shared random stream
        Symbolic random number generator used to draw random velocity and
        perform accept-reject move.
    positions: shared theano matrix
        Symbolic matrix whose rows are position vectors.
    energy_fn: python function
        Python function, operating on symbolic theano variables, used to
        compute the potential energy at a given position.
    stepsize: shared theano scalar
        Shared variable containing the stepsize to use for \( n\) steps' of HMC
        simulation steps.
    n_steps: integer
```
Number of HMC steps to perform before proposing a new position.

Returns
-------
rvall: boolean
    True if move is accepted, False otherwise
rval2: theano matrix
    Matrix whose rows contain the proposed "new position"

We start by sampling random velocities, using the provided shared RandomStream object. Velocities are sampled independently for each dimension and for each particle under simulation, yielding a $N \times D$ matrix.

```python
# sample random velocity
initial_vel = s_rng.normal(size=positions.shape)
```

Since we now have an initial position and velocity, we can now call the `simulate_dynamics` to obtain the proposal for the new state $\chi'$.

```python
# perform simulation of particles subject to Hamiltonian dynamics
final_pos, final_vel = simulate_dynamics(
    initial_pos=positions,
    initial_vel=initial_vel,
    stepsize=stepsize,
    n_steps=n_steps,
    energy_fn=energy_fn
)
```

We then accept/reject the proposed state based on the Metropolis algorithm.

```python
# accept/reject the proposed move based on the joint distribution
accept = metropolis_hastings_accept(
    energy_prev=hamiltonian(positions, initial_vel, energy_fn),
    energy_next=hamiltonian(final_pos, final_vel, energy_fn),
    s_rng=s_rng
)
```

where `metropolis_hastings_accept` and `hamiltonian` are helper functions, defined as follows.

```python
def metropolis_hastings_accept(energy_prev, energy_next, s_rng):
    """
    Performs a Metropolis-Hastings accept-reject move.
    """
    Parameters
    ----------
    energy_prev: theano vector
        Symbolic theano tensor which contains the energy associated with the
        configuration at time-step t.
    energy_next: theano vector
        Symbolic theano tensor which contains the energy associated with the
        proposed configuration at time-step t+1.
    s_rng: theano.tensor.shared_randomstreams.RandomStreams
```
Theano shared random stream object used to generate the random number used in proposal.

Returns
------
return: boolean
    True if move is accepted, False otherwise

```
ediff = energy_prev - energy_next
return (TT.exp(ediff) - s_rng.uniform(size=energy_prev.shape)) >= 0
```

def hamiltonian(pos, vel, energy_fn):
    
    Returns the Hamiltonian (sum of potential and kinetic energy) for the given velocity and position.

    Parameters
    ----------
    pos: theano matrix
        Symbolic matrix whose rows are position vectors.
    vel: theano matrix
        Symbolic matrix whose rows are velocity vectors.
    energy_fn: python function
        Python function, operating on symbolic theano variables, used to compute the potential energy at a given position.

    Returns
    ------
    return: theano vector
        Vector whose i-th entry is the Hamiltonian at position pos[i] and velocity vel[i].

    # assuming mass is 1
    return energy_fn(pos) + kinetic_energy(vel)

def kinetic_energy(vel):
    
    Returns the kinetic energy associated with the given velocity and mass of 1.

    Parameters
    ----------
    vel: theano matrix
        Symbolic matrix whose rows are velocity vectors.

    Returns
    ------
    return: theano vector
        Vector whose i-th entry is the kinetic entry associated with vel[i].

    #
    return 0.5 * (vel ** 2).sum(axis=1)
hmc_move finally returns the tuple \((\text{accept}, \text{final\_pos})\). \text{accept} is a symbolic boolean variable indicating whether or not the new state \text{final\_pos} should be used or not.

**hmc_updates**

The purpose of \text{hmc\_updates} is to generate the list of updates to perform, whenever our HMC sampling function is called. \text{hmc\_updates} thus receives as parameters, a series of shared variables to update \((\text{positions}, \text{stepsize} \text{ and } \text{avg\_acceptance\_rate})\), and the parameters required to compute their new state.

```python
def hmc_updates(positions, stepsize, avg_acceptance_rate, final_pos, accept, target_acceptance_rate, stepsize_inc, stepsize_dec, stepsize_min, stepsize_max, avg_acceptance_slowness):
    """This function is executed after \`n\_steps\` of HMC sampling (\`hmc\_move\` function). It creates the updates dictionary used by the \`simulate\` function. It takes care of updating: the position (if the move is accepted), the stepsize (to track a given target acceptance rate) and the average acceptance rate (computed as a moving average).

Parameters
----------
positions: shared variable, theano matrix
    Shared theano matrix whose rows contain the old position
stepsize: shared variable, theano scalar
    Shared theano scalar containing current step size
avg\_acceptance\_rate: shared variable, theano scalar
    Shared theano scalar containing the current average acceptance rate
final\_pos: shared variable, theano matrix
    Shared theano matrix whose rows contain the new position
accept: theano scalar
    Boolean-type variable representing whether or not the proposed HMC move should be accepted or not.
target_acceptance\_rate: float
    The stepsize is modified in order to track this target acceptance rate.
stepsize\_inc: float
    Amount by which to increment stepsize when acceptance rate is too high.
stepsize\_dec: float
    Amount by which to decrement stepsize when acceptance rate is too low.
stepsize\_min: float
    Lower-bound on \`stepsize\`.
stepsize\_max: float
    Upper-bound on \`stepsize\`.
avg\_acceptance\_slowness: float
    Average acceptance rate is computed as an exponential moving average. \((1-\text{avg\_acceptance\_slowness})\) is the weight given to the newest observation.

Returns
-------
rval1: dictionary-like
    A dictionary of updates to be used by the \`HMC\_Sampler\_simulate\` function.
```

11.2. Implementing HMC Using Theano

129
function. The updates target the position, stepsize and average acceptance rate.

```python
# POSITION UPDATES #
# broadcast `accept` scalar to tensor with the same dimensions as
# final_pos.
accept_matrix = accept.dimshuffle(0, *(('x',) * (final_pos.ndim - 1)))
# if accept is True, update to `final_pos` else stay put
new_positions = TT.switch(accept_matrix, final_pos, positions)
```

Using the above code, the dictionary `positions : new_positions` can be used to update the state of the sampler with either (1) the new state `final_pos` if `accept` is True, or (2) the old state if `accept` is False. This conditional assignment is performed by the `switch` op.

`switch` expects as its first argument, a boolean mask with the same broadcastable dimensions as the second and third argument. Since `accept` is scalar-valued, we must first use dimshuffle to transform it to a tensor with `final_pos.ndim` broadcastable dimensions (`accept_matrix`).

`hmc_updates` additionally implements an adaptive version of HMC, as implemented in the accompanying code to [Ranzato10]. We start by tracking the average acceptance rate of the HMC move proposals (across many simulations), using an exponential moving average with time constant \(1 - \text{avg\_acceptance\_slowness}\).

```python
# ACCEPT RATE UPDATES #
# perform exponential moving average
mean_dtype = theano.scalar.upcast(accept.dtype, avg_acceptance_rate.dtype)
new_acceptance_rate = TT.add(
    avg_acceptance_slowness * avg_acceptance_rate,
    (1.0 - avg_acceptance_slowness) * accept.mean(dtype=mean_dtype))
```

If the average acceptance rate is larger than the `target_acceptance_rate`, we increase the stepsize by a factor of `stepsize_inc` in order to increase the mixing rate of our chain. If the average acceptance rate is too low however, `stepsize` is decreased by a factor of `stepsize_dec`, yielding a more conservative mixing rate. The `clip` op allows us to maintain the stepsize in the range \([\text{stepsize\_min}, \text{stepsize\_max}]\).

```python
# STEPSIZE UPDATES #
# if acceptance rate is too low, our sampler is too "noisy" and we reduce
# the stepsize. If it is too high, our sampler is too conservative, we can
# get away with a larger stepsize (resulting in better mixing).
_new_stepsize = TT.switch(avg_acceptance_rate > target_acceptance_rate,
    stepsize * stepsize_inc, stepsize * stepsize_→dec)
# maintain stepsize in [stepsize_min, stepsize_max]
new_stepsize = TT.clip(_new_stepsize, stepsize_min, stepsize_max)
```

The final updates list is then returned.

```python
return [(positions, new_positions),
        (stepsize, new_stepsize),
```
HMC_sampler

We finally tie everything together using the `HMC_Sampler` class. Its main elements are:

- *new_from_shared_positions*: a constructor method which allocates various shared variables and strings together the calls to `hmc_move` and `hmc_updates`. It also builds the theano function `simulate`, whose sole purpose is to execute the updates generated by `hmc_updates`.

- *draw*: a convenience method which calls the Theano function `simulate` and returns a copy of the contents of the shared variable `self.positions`.

```python
class HMC_sampler(object):
    ""
    Convenience wrapper for performing Hybrid Monte Carlo (HMC). It creates the symbolical graph for performing an HMC simulation (using `hmc_move` and `hmc_updates`). The graph is then compiled into the `simulate` function, a theano function which runs the simulation and updates the required shared variables.

    Users should interface with the sampler thorough the `draw` function which advances the markov chain and returns the current sample by calling `simulate` and `get_position` in sequence.

    The hyper-parameters are the same as those used by Marc'Aurelio's 'train_mcRBM.py' file (available on his personal home page).
    ""

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    @classmethod
def new_from_shared_positions(self, shared_positions, energy_fn, initial_stepsize=0.01, target_acceptance_rate=.9, n_steps=20, stepsize_dec=0.98, stepsize_min=0.001, stepsize_max=0.25, stepsize_inc=1.02, avg_acceptance_slowness=0.9, seed=12345):
        ""
        :param shared_positions: theano ndarray shared var with many particle [initial] positions
```
:param energy_fn:
callable such that energy_fn(positions) returns theano vector of energies.
The len of this vector is the batchsize.

The sum of this energy vector must be differentiable (with
theano.tensor.grad) with respect to the positions for HMC
sampling to work.

```
# allocate shared variables
stepsize = sharedX(initial_stepsize, 'hmc_stepsize')
avg_acceptance_rate = sharedX(target_acceptance_rate,
                               'avg_acceptance_rate')
s_rng = theano.sandbox.rng_mrg.MRG_RandomStreams(seed)

# define graph for an `n_steps` HMC simulation
accept, final_pos = hmc_move(
    s_rng,
    shared_positions,
    energy_fn,
    stepsize,
    n_steps)

# define the dictionary of updates, to apply on every `simulate` call
simulate_updates = hmc_updates(
    shared_positions,
    stepsize,
    avg_acceptance_rate,
    final_pos=final_pos,
    accept=accept,
    stepsize_min=stepsize_min,
    stepsize_max=stepsize_max,
    stepsize_inc=stepsize_inc,
    stepsize_dec=stepsize_dec,
    target_acceptance_rate=target_acceptance_rate,
    avg_acceptance_slowness=avg_acceptance_slowness)

# compile theano function
simulate = function([], [], updates=simulate_updates)

# create HMC_sampler object with the following attributes ...
return cls(
    positions=shared_positions,
    stepsize=stepsize,
    stepsize_min=stepsize_min,
    stepsize_max=stepsize_max,
    avg_acceptance_rate=avg_acceptance_rate,
    target_acceptance_rate=target_acceptance_rate,
    s_rng=s_rng,
    updates=simulate_updates,
    simulate=simulate)
11.3 Testing our Sampler

We test our implementation of HMC by sampling from a multi-variate Gaussian distribution. We start by generating a random mean vector \( \mu \) and covariance matrix \( \text{cov} \), which allows us to define the energy function of the corresponding Gaussian distribution: \text{gaussian\_energy}. We then initialize the state of the sampler by allocating a \text{position} shared variable. It is passed to the constructor of \text{HMC\_sampler} along with our target energy function.

Following a burn-in period, we then generate a large number of samples and compare the empirical mean and covariance matrix to their true values.

```python
def draw(self, **kwargs):
    ""
    Returns a new position obtained after \text{`n\_steps\'} of HMC simulation.
    Parameters
    *******
    kwargs: dictionary
    The `kwargs` dictionary is passed to the shared variable (self.positions) `get_value()` function. For example, to avoid copying the shared variable value, consider passing `borrow=True`.
    Returns
    *******
    rval: numpy matrix
    Numpy matrix whose of dimensions similar to \text{`initial\_position`}.
    ""
    self.simulate()
    return self.positions.get_value(borrow=False)

def sampler_on_nd_gaussian(sampler_cls, burnin, n_samples, dim=10):
    batchsize = 3
    rng = numpy.random.RandomState(123)
    # Define a covariance and mu for a gaussian
    mu = numpy.array(rng.rand(dim) * 10, dtype=theano.config.floatX)
    cov = numpy.array(rng.rand(dim, dim), dtype=theano.config.floatX)
    cov = (cov + cov.T) / 2.
    cov[numpy.arange(dim), numpy.arange(dim)] = 1.0
    cov_inv = numpy.linalg.inv(cov)
    # Define energy function for a multi-variate Gaussian
    def gaussian_energy(x):
        return 0.5 * (theano.tensor.dot((x - mu), cov_inv) *
                    (x - mu)).sum(axis=1)
    # Declared shared random variable for positions
    position = rng.randn(batchsize, dim).astype(theano.config.floatX)
    position = theano.shared(position)
```

11.3. Testing our Sampler
# Create HMC sampler
sampler = sampler_cls(position, gaussian_energy,
    initial_stepsize=1e-3, stepsize_max=0.5)

# Start with a burn-in process
garbage = [sampler.draw() for r in range(burnin)] # burn-in Draw
# \`n_samples\`: result is a 3D tensor of dim [n_samples, batchsize, 
# dim]
_samples = numpy.asarray([sampler.draw() for r in range(n_samples)])
# Flatten to [n_samples * batchsize, dim]
samples = _samples.T.reshape(dim, -1).T

print('****** TARGET VALUES ******
print('target mean:', mu)
print('target cov:

', cov)

print('****** EMPIRICAL MEAN/COV USING HMC ******
print('empirical mean: ', samples.mean(axis=0))
print('empirical_cov:

', numpy.cov(samples.T))

print('****** HMC INTERNALS ******
print('final stepsize', sampler.stepsize.get_value())
print('final acceptance_rate', sampler.avg_acceptance_rate.get_value())

return sampler

def test_hmc():
    sampler = sampler_on_nd_gaussian(HMC_sampler.new_from_shared_positions,
        burnin=1000, n_samples=1000, dim=5)
    assert abs(sampler.avg_acceptance_rate.get_value() -
        sampler.target_acceptance_rate) < .1
    assert sampler.stepsize.get_value() >= sampler.stepsize_min
    assert sampler.stepsize.get_value() <= sampler.stepsize_max

The above code can be run using the command: “nositests -s code/hmc/test_hmc.py”. The output is as follows:

[desjagui@atchoum hmc]$ python test_hmc.py

****** TARGET VALUES ******
target mean: [ 6.96469186 2.86139335 2.26851454 5.51314769 7.1946897 ]
target cov:
[[ 1. 0.66197111 0.71141257 0.55766643 0.35753822]
 [ 0.66197111 1. 0.31053199 0.45455485 0.37991646]
 [ 0.71141257 0.31053199 1. 0.62800335 0.38004541]
 [ 0.55766643 0.45455485 0.62800335 1. 0.50807871]
 [ 0.35753822 0.37991646 0.38004541 0.50807871 1. ]]

****** EMPIRICAL MEAN/COV USING HMC ******
empirical mean: [ 6.94155164 2.81526039 2.26301715 5.46536853 7.19414496]
empirical_cov:
As can be seen above, the samples generated by our HMC sampler yield an empirical mean and covariance matrix, which are very close to the true underlying parameters. The adaptive algorithm also seemed to work well as the final acceptance rate is close to our target of 0.9.

### 11.4 References
12.1 Summary

In this tutorial, you will learn how to:

• learn Word Embeddings
• using Recurrent Neural Networks architectures
• with Context Windows

in order to perform Semantic Parsing / Slot-Filling (Spoken Language Understanding)

12.2 Code - Citations - Contact

12.2.1 Code

Directly running experiments is also possible using this github repository.

12.2.2 Papers

If you use this tutorial, cite the following papers:


• [pdf] Gokhan Tur, Dilek Hakkani-Tur and Larry Heck. What is left to be understood in ATIS?


Thank you!

12.2.3 Contact

Please email to Grégoire Mesnil (first-add-a-dot-last-add-at-gmail-add-a-dot-com) for any problem report or feedback. We will be glad to hear from you.

12.3 Task

The Slot-Filling (Spoken Language Understanding) consists in assigning a label to each word given a sentence. It’s a classification task.

12.4 Dataset

An old and small benchmark for this task is the ATIS (Airline Travel Information System) dataset collected by DARPA. Here is a sentence (or utterance) example using the Inside Outside Beginning (IOB) representation.

<table>
<thead>
<tr>
<th>Input (words)</th>
<th>show</th>
<th>flights</th>
<th>from</th>
<th>Boston</th>
<th>to</th>
<th>New</th>
<th>York</th>
<th>today</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output (labels)</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>B-dept</td>
<td>O</td>
<td>B-arr</td>
<td>I-arr</td>
<td>B-date</td>
</tr>
</tbody>
</table>

The ATIS official split contains 4,978/893 sentences for a total of 56,590/9,198 words (average sentence length is 15) in the train/test set. The number of classes (different slots) is 128 including the O label (NULL).

As Microsoft Research people, we deal with unseen words in the test set by marking any words with only one single occurrence in the training set as <UNK> and use this token to represent those unseen words in the test set. As Ronan Collobert and colleagues, we converted sequences of numbers with the string DIGIT i.e. 1984 is converted to DIGITDIGITDIGITDIGIT.

We split the official train set into a training and validation set that contain respectively 80% and 20% of the official training sentences. Significant performance improvement difference has to be greater than 0.6% in F1 measure at the 95% level due to the small size of the dataset. For evaluation purpose, experiments have to report the following metrics:

• Precision
• Recall
• F1 score

We will use the conlleval PERL script to measure the performance of our models.

12.5 Recurrent Neural Network Model

12.5.1 Raw input encoding

A token corresponds to a word. Each token in the ATIS vocabulary is associated to an index. Each sentence is a array of indexes (int32). Then, each set (train, valid, test) is a list of arrays of indexes. A python dictionary is defined for mapping the space of indexes to the space of words.

```python
>>> sentence
array([383, 189, 13, 193, 208, 307, 195, 502, 260, 539, 7, 60, 72, 8, 350, 384], dtype=int32)
```

```python
>>> map(lambda x: index2word[x], sentence)
['please', 'find', 'a', 'flight', 'from', 'miami', 'florida', 'to', 'las', 'vegas', '<UNK>', 'arriving', 'before', 'DIGIT', "o'clock →", 'pm']
```

Same thing for labels corresponding to this particular sentence.

```python
>>> labels
array([126, 126, 126, 126, 126, 48, 50, 126, 78, 123, 81, 126, 15, 14, 89, 89], dtype=int32)
```

```python
>>> map(lambda x: index2label[x], labels)
['O', 'O', 'O', 'O', 'O', 'B-fromloc.city_name', 'B-fromloc.state_name', 'O', 'B-toloc.city_name', 'I-toloc.city_name', 'B-toloc.state_name', 'O', 'B-arrive_time.time_relative', 'B-arrive_time.time', 'I-arrive_time.time', 'I-arrive_time.time']
```

12.5.2 Context window

Given a sentence i.e. an array of indexes, and a window size i.e. 1,3,5,..., we need to convert each word in the sentence to a context window surrounding this particular word. In details, we have:

```python
def contextwin(l, win):
    ...
    win :: int corresponding to the size of the window
    given a list of indexes composing a sentence
    
    l :: array containing the word indexes
    
    it will return a list of list of indexes corresponding to context windows surrounding each word in the sentence
    ...
    assert (win % 2) == 1
    assert win >= 1
    l = list(l)
```
lpadded = win // 2 * [-1] + 1 + win // 2 * [-1]
out = [lpadded[i:(i + win)] for i in range(len(l))]

assert len(out) == len(l)
return out

The index -1 corresponds to the PADDING index we insert at the beginning/end of the sentence.

Here is a sample:

```python
>>> x
array([0, 1, 2, 3, 4], dtype=int32)
>>> contextwin(x, 3)
[[-1, 0, 1],
 [ 0, 1, 2],
 [ 1, 2, 3],
 [ 2, 3, 4],
 [ 3, 4,-1]]
>>> contextwin(x, 7)
[[-1, -1, -1, 0, 1, 2, 3],
 [-1, -1, 0, 1, 2, 3, 4],
 [-1, 0, 1, 2, 3, 4,-1],
 [ 0, 1, 2, 3, 4,-1,-1],
 [ 1, 2, 3, 4,-1,-1,-1]]
```

To summarize, we started with an array of indexes and ended with a matrix of indexes. Each line corresponds to the context window surrounding this word.

### 12.5.3 Word embeddings

Once we have the sentence converted to context windows i.e. a matrix of indexes, we have to associate these indexes to the embeddings (real-valued vector associated to each word). Using Theano, it gives:

```python
import theano, numpy
from theano import tensor as T

# nv :: size of our vocabulary
# de :: dimension of the embedding space
# cs :: context window size
nv, de, cs = 1000, 50, 5

embeddings = theano.shared(0.2 * numpy.random.uniform(-1.0, 1.0, (nv+1, de)).astype(theano.config.floatX))  # add one for PADDING at the end

idxs = T.imatrix()  # as many columns as words in the context window and as many lines as words in the sentence

x = self.emb[idxs].reshape(((idxs.shape[0], de*cs))

The x symbolic variable corresponds to a matrix of shape (number of words in the sentences, dimension of the embedding space X context window size).
Let's compile a theano function to do so

```python
>>> sample
array([0, 1, 2, 3, 4], dtype=int32)
>>> csample = contextwin(sample, 7)
[-1, -1, -1, 0, 1, 2, 3],
[-1, -1, 0, 1, 2, 3, 4],
[-1, 0, 1, 2, 3, 4,-1],
[ 0, 1, 2, 3, 4,-1,-1],
[ 1, 2, 3, 4,-1,-1,-1])
>>> f = theano.function(inputs=[idxs], outputs=x)
>> f(csample)
anarray([[ 0.08088442, 0.08458307, 0.05064092, ..., 0.06876887,
  -0.06648078, -0.15192257],
 [-0.08088442, 0.08458307, 0.05064092, ..., 0.11192625,
  0.08745284, 0.04381778],
[-0.08088442, 0.08458307, 0.05064092, ..., -0.00937143,
  0.10804889, 0.1247109 ],
[ 0.11038255, -0.10563177, -0.18760249, ..., -0.00937143,
  0.10804889, 0.1247109 ],
[ 0.18738101, 0.14727569, -0.069544, ..., -0.00937143,
  0.10804889, 0.1247109 ]]), dtype=float32)
>>> f(csample).shape
(5, 350)
```

We now have a sequence (of length 5 which is corresponds to the length of the sentence) of context window word embeddings which is easy to feed to a simple recurrent neural network to iterate with.

### 12.5.4 Elman recurrent neural network

The followin (Elman) recurrent neural network (E-RNN) takes as input the current input (time \( t \)) and the previous hiddent state (time \( t-1 \)). Then it iterates.

In the previous section, we processed the input to fit this sequential/temporal structure. It consists in a matrix where the row 0 corresponds to the time step \( t=0 \), the row 1 corresponds to the time step \( t=1 \), etc.

The parameters of the E-RNN to be learned are:

- the word embeddings (real-valued matrix)
- the initial hidden state (real-value vector)
- two matrices for the linear projection of the input \( t \) and the previous hidden layer state \( t-1 \)
- (optional) bias. **Recommendation:** don’t use it.
- softmax classification layer on top

The hyperparameters define the whole architecture:

- dimension of the word embedding
- size of the vocabulary
- number of hidden units
• number of classes
• random seed + way to initialize the model

It gives the following code:

```python
class RNNSLU(object):
    ''' elman neural net model '''
    def __init__(self, nh, nc, ne, de, cs):
        '''
        nh :: dimension of the hidden layer
        nc :: number of classes
        ne :: number of word embeddings in the vocabulary
        de :: dimension of the word embeddings
        cs :: word window context size
        '''
        # parameters of the model
        self.emb = theano.shared(name='embeddings',
                                 value=0.2 * numpy.random.uniform(-1.0, 1.0, (ne+1, de))
                                 # add one for padding at the end
                                 .astype(theano.config.floatX))
        self.wx = theano.shared(name='wx',
                                value=0.2 * numpy.random.uniform(-1.0, 1.0, (de * cs, nh))
                                .astype(theano.config.floatX))
        self.wh = theano.shared(name='wh',
                                value=0.2 * numpy.random.uniform(-1.0, 1.0, (nh, nh))
                                .astype(theano.config.floatX))
        self.w = theano.shared(name='w',
                               value=0.2 * numpy.random.uniform(-1.0, 1.0, (nh, nc))
                               .astype(theano.config.floatX))
        self.bh = theano.shared(name='bh',
                                value=numpy.zeros(nh, dtype=theano.config.floatX))
        self.b = theano.shared(name='b',
                               value=numpy.zeros(nc, dtype=theano.config.floatX))
        self.h0 = theano.shared(name='h0',
                                value=numpy.zeros(nh, dtype=theano.config.floatX))

        # bundle
        self.params = [self.emb, self.wx, self.wh, self.w, self.bh, self.b, self.h0]
```

Then we integrate the way to build the input from the embedding matrix:

```python
idxs = T.imatrix()
x = self.emb[idxs].reshape((idxs.shape[0], de*cs))
y_sentence = T.ivector('y_sentence')  # labels
```
We use the scan operator to construct the recursion, works like a charm:

```python
def recurrence(x_t, h_tm1):
    h_t = T.nnet.sigmoid(T.dot(x_t, self.wx)
                         + T.dot(h_tm1, self.wh) + self.bh)
    s_t = T.nnet.softmax(T.dot(h_t, self.w) + self.b)
    return [h_t, s_t]

[h, s], _ = theano.scan(fn=recurrence,
                        sequences=x,
                        outputs_info=[self.h0, None],
                        n_steps=x.shape[0])

p_y_given_x_sentence = s[:, 0, :]
y_pred = T.argmax(p_y_given_x_sentence, axis=1)
```

Theano will then compute all the gradients automatically to maximize the log-likelihood:

```python
lr = T.scalar('lr')

sentence_nll = -T.mean(T.log(p_y_given_x_sentence)
                        [T.arange(x.shape[0]), y_sentence])
sentence_gradients = T.grad(sentence_nll, self.params)
sentence_updates = OrderedDict((p, p - lr*g)
                                for p, g in
                                zip(self.params, sentence_gradients))
```

Next compile those functions:

```python
self.classify = theano.function(inputs=[idxs], outputs=y_pred)
self.sentence_train = theano.function(inputs=[idxs, y_sentence, lr],
                                       outputs=sentence_nll,
                                       updates=sentence_updates)
```

We keep the word embeddings on the unit sphere by normalizing them after each update:

```python
self.normalize = theano.function(inputs=[],
                                 updates={self.emb:
                                      self.emb /
                                      T.sqrt((self.emb**2)
                                          .sum(axis=1))
                                      .dimshuffle(0, 'x')))
```

And that’s it!

### 12.6 Evaluation

With the previous defined functions, you can compare the predicted labels with the true labels and compute some metrics. In this repo, we build a wrapper around the conlleval PERL script. It’s not trivial to compute those metrics due to the Inside Outside Beginning (IOB) representation i.e. a prediction is considered correct
if the word-beginning and the word-inside and the word-outside predictions are all correct. Note that the extension is txt and you will have to change it to pl.

12.7 Training

12.7.1 Updates

For stochastic gradient descent (SGD) update, we consider the whole sentence as a mini-batch and perform one update per sentence. It is possible to perform a pure SGD (contrary to mini-batch) where the update is done on only one single word at a time.

After each iteration/update, we normalize the word embeddings to keep them on a unit sphere.

12.7.2 Stopping Criterion

Early-stopping on a validation set is our regularization technique: the training is run for a given number of epochs (a single pass through the whole dataset) and keep the best model along with respect to the F1 score computed on the validation set after each epoch.

12.7.3 Hyper-Parameter Selection

Although there is interesting research/code on the topic of automatic hyper-parameter selection, we use the KISS random search.

The following intervals can give you some starting point:

- learning rate : uniform([0.05,0.01])
- window size : random value from {3,...,19}
- number of hidden units : random value from {100,200}
- embedding dimension : random value from {50,100}

12.8 Running the Code

After downloading the data using download.sh, the user can then run the code by calling:

```python
code/rnnslu.py
('NEW BEST: epoch', 25, 'valid F1', 96.84, 'best test F1', 93.79)
[learning] epoch 26 >> 100.00% completed in 28.76 (sec) <<
[learning] epoch 27 >> 100.00% completed in 28.76 (sec) <<
...
('BEST RESULT: epoch', 57, 'valid F1', 97.23, 'best test F1', 94.2, 'with the model', 'rnnslu')
```
12.8.1 Timing

Running experiments on ATIS using this repository will run one epoch in less than 40 seconds on i7 CPU 950 @ 3.07GHz using less than 200 Mo of RAM:

```plaintext
[learning] epoch 0 >> 100.00% completed in 34.48 (sec) <<
```

After a few epochs, you obtain decent performance 94.48 % of F1 score:

```plaintext
NEW BEST: epoch 28 valid F1 96.61 best test F1 94.19
NEW BEST: epoch 29 valid F1 96.63 best test F1 94.42
[learning] epoch 30 >> 100.00% completed in 35.04 (sec) <<
[learning] epoch 31 >> 100.00% completed in 34.80 (sec) <<
[...]  
NEW BEST: epoch 40 valid F1 97.25 best test F1 94.34
[learning] epoch 41 >> 100.00% completed in 35.18 (sec) <<
NEW BEST: epoch 42 valid F1 97.33 best test F1 94.48
[learning] epoch 43 >> 100.00% completed in 35.39 (sec) <<
[learning] epoch 44 >> 100.00% completed in 35.31 (sec) <<
[...]  
```

12.8.2 Word Embedding Nearest Neighbors

We can check the k-nearest neighbors of the learned embeddings. L2 and cosine distance gave the same results so we plot them for the cosine distance.

<table>
<thead>
<tr>
<th>atlanta</th>
<th>back</th>
<th>ap80</th>
<th>but</th>
<th>aircraft</th>
<th>business</th>
<th>a</th>
<th>august</th>
<th>actually</th>
<th>cheap</th>
</tr>
</thead>
<tbody>
<tr>
<td>phoenix</td>
<td>live</td>
<td>ap57</td>
<td>if</td>
<td>plane</td>
<td>coach</td>
<td>people</td>
<td>september</td>
<td>provide</td>
<td>weekday</td>
</tr>
<tr>
<td>denver</td>
<td>lives</td>
<td>ap</td>
<td>up</td>
<td>service</td>
<td>first</td>
<td>do</td>
<td>january</td>
<td>prices</td>
<td>weekdays</td>
</tr>
<tr>
<td>tacoma</td>
<td>both</td>
<td>connect</td>
<td>a</td>
<td>airplane</td>
<td>fourth</td>
<td>but</td>
<td>june</td>
<td>stop</td>
<td>am</td>
</tr>
<tr>
<td>columbus</td>
<td>how</td>
<td>tomorrow</td>
<td>now</td>
<td>seating</td>
<td>thrift</td>
<td>numbers</td>
<td>december</td>
<td>number</td>
<td>early</td>
</tr>
<tr>
<td>seattle</td>
<td>me</td>
<td>before</td>
<td>amount</td>
<td>stand</td>
<td>tenth</td>
<td>abbreviation</td>
<td>november</td>
<td>flight</td>
<td>sfo</td>
</tr>
<tr>
<td>minneapolis</td>
<td>out</td>
<td>earliest</td>
<td>more</td>
<td>that</td>
<td>second</td>
<td>if</td>
<td>april</td>
<td>there</td>
<td>milwaukee</td>
</tr>
<tr>
<td>pittsburgh</td>
<td>other</td>
<td>connect</td>
<td>abbreviation</td>
<td>on</td>
<td>fifth</td>
<td>up</td>
<td>july</td>
<td>serving</td>
<td>jfk</td>
</tr>
<tr>
<td>ontario</td>
<td>plane</td>
<td>thrift</td>
<td>restrictions</td>
<td>turbo-prop</td>
<td>third</td>
<td>serve</td>
<td>jfk</td>
<td>thank</td>
<td>shortest</td>
</tr>
<tr>
<td>montreal</td>
<td>service</td>
<td>coach</td>
<td>mean</td>
<td>mean</td>
<td>twelfth</td>
<td>database</td>
<td>october</td>
<td>ticket</td>
<td>bwi</td>
</tr>
<tr>
<td>philadelphia</td>
<td>fare</td>
<td>today</td>
<td>interested</td>
<td>amount</td>
<td>sixth</td>
<td>passengers</td>
<td>may</td>
<td>are</td>
<td>lastest</td>
</tr>
</tbody>
</table>
As you can judge, the limited size of the vocabulary (about 500 words) gives us mitigated performance. According to human judgement: some are good, some are bad.
13.1 Summary

This tutorial aims to provide an example of how a Recurrent Neural Network (RNN) using the Long Short Term Memory (LSTM) architecture can be implemented using Theano. In this tutorial, this model is used to perform sentiment analysis on movie reviews from the Large Movie Review Dataset, sometimes known as the IMDB dataset.

In this task, given a movie review, the model attempts to predict whether it is positive or negative. This is a binary classification task.

13.2 Data

As previously mentioned, the provided scripts are used to train a LSTM recurrent neural network on the Large Movie Review Dataset dataset.

While the dataset is public, in this tutorial we provide a copy of the dataset that has previously been pre-processed according to the needs of this LSTM implementation. Running the code provided in this tutorial will automatically download the data to the local directory. In order to use your own data, please use a (preprocessing script) provided as a part of this tutorial.

Once the model is trained, you can test it with your own corpus using the word-index dictionary (imdb.dict.pkl.gz) provided as a part of this tutorial.
13.3 Model

13.3.1 LSTM

In a traditional recurrent neural network, during the gradient back-propagation phase, the gradient signal can end up being multiplied a large number of times (as many as the number of timesteps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. This means that, the magnitude of weights in the transition matrix can have a strong impact on the learning process.

If the weights in this matrix are small (or, more formally, if the leading eigenvalue of the weight matrix is smaller than 1.0), it can lead to a situation called vanishing gradients where the gradient signal gets so small that learning either becomes very slow or stops working altogether. It can also make more difficult the task of learning long-term dependencies in the data. Conversely, if the weights in this matrix are large (or, again, more formally, if the leading eigenvalue of the weight matrix is larger than 1.0), it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as exploding gradients.

These issues are the main motivation behind the LSTM model which introduces a new structure called a memory cell (see Figure 1 below). A memory cell is composed of four main elements: an input gate, a neuron with a self-recurrent connection (a connection to itself), a forget gate and an output gate. The self-recurrent connection has a weight of 1.0 and ensures that, barring any outside interference, the state of a memory cell can remain constant from one timestep to another. The gates serve to modulate the interactions between the memory cell itself and its environment. The input gate can allow incoming signal to alter the state of the memory cell or block it. On the other hand, the output gate can allow the state of the memory cell to have an effect on other neurons or prevent it. Finally, the forget gate can modulate the memory cell’s self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.

![Figure 1: Illustration of an LSTM memory cell.](image)

The equations below describe how a layer of memory cells is updated at every timestep $t$. In these equations:

- $x_t$ is the input to the memory cell layer at time $t$
• $W_i, W_f, W_c, W_o, U_i, U_f, U_c, U_o$ and $V_o$ are weight matrices
• $b_i, b_f, b_c$ and $b_o$ are bias vectors

First, we compute the values for $i_t$, the input gate, and $\tilde{C}_t$ the candidate value for the states of the memory cells at time $t$:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

(13.1)

$$\tilde{C}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$

(13.2)

Second, we compute the value for $f_t$, the activation of the memory cells’ forget gates at time $t$:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

(13.3)

Given the value of the input gate activation $i_t$, the forget gate activation $f_t$ and the candidate state value $\tilde{C}_t$, we can compute $C_t$ the memory cells’ new state at time $t$:

$$C_t = i_t \ast \tilde{C}_t + f_t \ast C_{t-1}$$

(13.4)

With the new state of the memory cells, we can compute the value of their output gates and, subsequently, their outputs:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$

(13.5)

$$h_t = o_t \ast tanh(C_t)$$

(13.6)

13.3.2 Our model

The model we used in this tutorial is a variation of the standard LSTM model. In this variant, the activation of a cell’s output gate does not depend on the memory cell’s state $C_t$. This allows us to perform part of the computation more efficiently (see the implementation note, below, for details). This means that, in the variant we have implemented, there is no matrix $V_o$ and equation (13.5) is replaced by equation (13.7):

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

(13.7)

Our model is composed of a single LSTM layer followed by an average pooling and a logistic regression layer as illustrated in Figure 2 below. Thus, from an input sequence $x_0, x_1, x_2, ..., x_n$, the memory cells in the LSTM layer will produce a representation sequence $h_0, h_1, h_2, ..., h_n$. This representation sequence is then averaged over all timesteps resulting in representation $h$. Finally, this representation is fed to a logistic regression layer whose target is the class label associated with the input sequence.

Implementation note: In the code included this tutorial, the equations (13.1), (13.2), (13.3) and (13.7) are performed in parallel to make the computation more efficient. This is possible because none of these equations rely on a result produced by the other ones. It is achieved by concatenating the four matrices $W_*$ into a single weight matrix $W$ and performing the same concatenation on the weight matrices $U_*$ to produce
Fig. 13.2: Figure 2: Illustration of the model used in this tutorial. It is composed of a single LSTM layer followed by mean pooling over time and logistic regression.
the matrix $U$ and the bias vectors $b_*$ to produce the vector $b$. Then, the pre-nonlinearity activations can be computed with:

$$z = W x_t + U h_{t-1} + b$$

The result is then sliced to obtain the pre-nonlinearity activations for $i$, $f$, $\tilde{C}_t$, and $o$ and the non-linearities are then applied independently for each.

### 13.4 Code - Citations - Contact

#### 13.4.1 Code

The LSTM implementation can be found in the two following files:

- **lstm.py**: Main script. Defines and train the model.
- **imdb.py**: Secondary script. Handles the loading and preprocessing of the IMDB dataset.

After downloading both scripts and putting both in the same folder, the user can run the code by calling:

```python
THEANO_FLAGS="floatX=float32" python lstm.py
```

The script will automatically download the data and decompress it.

**Note**: The provided code supports the Stochastic Gradient Descent (SGD), AdaDelta and RMSProp optimization methods. You are advised to use AdaDelta or RMSProp because SGD appears to performs poorly on this task with this particular model.

#### 13.4.2 Papers

If you use this tutorial, please cite the following papers.

Introduction of the LSTM model:


Addition of the forget gate to the LSTM model:


More recent LSTM paper:


Papers related to Theano:


Thank you!

13.4.3 Contact

Please email Pierre Luc Carrier or Kyunghyun Cho for any problem report or feedback. We will be glad to hear from you.

13.5 References


Modeling and generating sequences of polyphonic music with the RNN-RBM

Note: This tutorial demonstrates a basic implementation of the RNN-RBM as described in [BoulangerLewandowski12] (pdf). We assume the reader is familiar with recurrent neural networks using the scan op and restricted Boltzmann machines (RBM).

Note: The code for this section is available for download here: rnnrbm.py.

You will need the modified Python MIDI package (GPL license) in your $PYTHONPATH or in the working directory in order to convert MIDI files to and from piano-rolls. The script also assumes that the content of the Nottingham Database of folk tunes has been extracted in the ../data directory. Alternative MIDI datasets are available here.

Note that both dependencies above can be setup automatically by running the download.sh script in the ../data directory of the Deep Learning Tutorials repository.

Caution: Need Theano 0.6 or more recent.

14.1 The RNN-RBM

The RNN-RBM is an energy-based model for density estimation of temporal sequences, where the feature vector \( v^{(t)} \) at time step \( t \) may be high-dimensional. It allows to describe multimodal conditional distributions of \( v^{(t)} | \mathcal{A}^{(t)} \), where \( \mathcal{A}^{(t)} \equiv \{ v^{(\tau)} | \tau < t \} \) denotes the sequence history at time \( t \), via a series of conditional
Deep Learning Tutorial, Release 0.1

RBMs (one a each time step) whose parameters $b^{(t)}_v, b^{(t)}_h$ depend on the output of a deterministic RNN with hidden units $u^{(t)}$:

$$b^{(t)}_v = b_v + W_{uv} u^{(t-1)}$$

(14.1)

$$b^{(t)}_h = b_h + W_{uh} u^{(t-1)}$$

(14.2)

and the single-layer RNN recurrence relation is defined by:

$$u^{(t)} = \tanh(b_u + W_{uu} u^{(t-1)} + W_{vu} v^{(t)})$$

(14.3)

The resulting model is unrolled in time in the following figure:

The overall probability distribution is given by the sum over the $T$ time steps in a given sequence:

$$P(\{v^{(t)}\}) = \sum_{t=1}^{T} P(v^{(t)}|A^{(t)})$$

(14.4)

where the right-hand side multiplicand is the marginalized probability of the $t^{th}$ RBM.

Note that for clarity of the implementation, contrarily to [BoulangerLewandowski12], we use the obvious naming convention for weight matrices and we use $u^{(t)}$ instead of $h^{(t)}$ for the recurrent hidden units.

14.2 Implementation

We wish to construct two Theano functions: one to train the RNN-RBM, and one to generate sample sequences from it.

For training, i.e. given $\{v^{(t)}\}$, the RNN hidden state $\{u^{(t)}\}$ and the associated $\{b^{(t)}_v, b^{(t)}_h\}$ parameters are deterministic and can be readily computed for each training sequence. A stochastic gradient descent (SGD)
update on the parameters can then be estimated via contrastive divergence (CD) on the individual time steps of a sequence in the same way that individual training examples are treated in a mini-batch for regular RBMs.

Sequence generation is similar except that the $v^{(t)}$ must be sampled sequentially at each time step with a separate (non-batch) Gibbs chain before being passed down to the recurrence and the sequence history.

### 14.2.1 The RBM layer

The `build_rbm` function shown below builds a Gibbs chain from an input mini-batch (a binary matrix) via the CD approximation. Note that it also supports a single frame (a binary vector) in the non-batch case.

```python
def build_rbm(v, W, bv, bh, k):
    '''Construct a k-step Gibbs chain starting at v for an RBM.

    v : Theano vector or matrix
       If a matrix, multiple chains will be run in parallel (batch).
    W : Theano matrix
       Weight matrix of the RBM.
    bv : Theano vector
       Visible bias vector of the RBM.
    bh : Theano vector
       Hidden bias vector of the RBM.
    k : scalar or Theano scalar
       Length of the Gibbs chain.

    Return a (v_sample, cost, monitor, updates) tuple:
    v_sample : Theano vector or matrix with the same shape as `v`
               Corresponds to the generated sample(s).
    cost : Theano scalar
           Expression whose gradient with respect to W, bv, bh is the CD-k
           approximation to the log-likelihood of `v` (training example) under
           the RBM. The cost is averaged in the batch case.
    monitor: Theano scalar
             Pseudo log-likelihood (also averaged in the batch case).
    updates: dictionary of Theano variable -> Theano variable
             The `updates` object returned by scan.''

    def gibbs_step(v):
        mean_h = T.nnet.sigmoid(T.dot(v, W) + bh)
        h = rng.binomial(size=mean_h.shape, n=1, p=mean_h,
                         dtype=theano.config.floatX)
        mean_v = T.nnet.sigmoid(T.dot(h, W.T) + bv)
        v = rng.binomial(size=mean_v.shape, n=1, p=mean_v,
                         dtype=theano.config.floatX)
        return mean_v, v

    chain, updates = theano.scan(lambda v: gibbs_step(v)[1], outputs_info=[v],
                                 n_steps=k)
    v_sample = chain[-1]
```

14.2. Implementation
mean_v = gibbs_step(v_sample)[0]
monitor = T.xlogx.xlogy0(v, mean_v) + T.xlogx.xlogy0(1 - v, 1 - mean_v)
monitor = monitor.sum() / v.shape[0]

def free_energy(v):
    return -(v * bv).sum() - T.log(1 + T.exp(T.dot(v, W) + bh)).sum()
cost = (free_energy(v) - free_energy(v_sample)) / v.shape[0]
return v_sample, cost, monitor, updates

14.2.2 The RNN layer

The build_rnnrbm function defines the RNN recurrence relation to obtain the RBM parameters; the recurrence function is flexible enough to serve both in the training scenario where \( v^{(t)} \) is given and the “batch” RBM is constructed at the end on the whole sequence at once, and in the generation scenario where \( v^{(t)} \) is sampled separately at each time step using the Gibbs chain defined above.

def build_rnnrbm(n_visible, n_hidden, n_hidden_recurrent):
    '''Construct a symbolic RNN-RBM and initialize parameters.

    n_visible : integer
        Number of visible units.
    n_hidden : integer
        Number of hidden units of the conditional RBMs.
    n_hidden_recurrent : integer
        Number of hidden units of the RNN.

    Return a (v, v_sample, cost, monitor, params, updates_train, v_t, updates_generate) tuple:

    v : Theano matrix
        Symbolic variable holding an input sequence (used during training)
v_sample : Theano matrix
        Symbolic variable holding the negative particles for CD log-likelihood
        gradient estimation (used during training)
cost : Theano scalar
        Expression whose gradient (considering v_sample constant) corresponds
        to the LL gradient of the RNN-RBM (used during training)
monitor : Theano scalar
        Frame-level pseudo-likelihood (useful for monitoring during training)
params : tuple of Theano shared variables
        The parameters of the model to be optimized during training.
updates_train : dictionary of Theano variable -> Theano variable
        Update object that should be passed to theano.function when compiling
        the training function.
    v_t : Theano matrix
        Symbolic variable holding a generated sequence (used during sampling)
    updates_generate : dictionary of Theano variable -> Theano variable
        Update object that should be passed to theano.function when compiling
        the generation function.'''
\[ W = \text{shared_normal}(\text{n\_visible}, \text{n\_hidden}, 0.01) \]
\[ bv = \text{shared_zeros}(\text{n\_visible}) \]
\[ bh = \text{shared_zeros}(\text{n\_hidden}) \]
\[ W_{uh} = \text{shared_normal}(\text{n\_hidden\_recurrent}, \text{n\_hidden}, 0.0001) \]
\[ W_{uv} = \text{shared_normal}(\text{n\_hidden\_recurrent}, \text{n\_visible}, 0.0001) \]
\[ W_{vu} = \text{shared_normal}(\text{n\_visible}, \text{n\_hidden\_recurrent}, 0.0001) \]
\[ W_{uu} = \text{shared_normal}(\text{n\_hidden\_recurrent}, \text{n\_hidden\_recurrent}, 0.0001) \]
\[ bu = \text{shared_zeros}(\text{n\_hidden\_recurrent}) \]

params = \( W, bv, bh, W_{uh}, W_{uv}, W_{vu}, W_{uu}, bu \)  # learned parameters as shared variables

\( v = \text{T.matrix}() \)  # a training sequence
\( u0 = \text{T.zeros}((\text{n\_hidden\_recurrent},)) \)  # initial value for the RNN hidden units

# If `v_t` is given, deterministic recurrence to compute the variable
# biases bv_t, bh_t at each time step. If `v_t` is None, same recurrence
# but with a separate Gibbs chain at each time step to sample (generate)
# from the RNN-RBM. The resulting sample v_t is returned in order to be
# passed down to the sequence history.

def recurrence(v_t, u_tm1):
    bv_t = bv + T.dot(u_tm1, W_{uv})
    bh_t = bh + T.dot(u_tm1, W_{uh})
    generate = v_t
    if generate:
        v_t, _, _, updates = \text{build\_rbm}(\text{T.zeros}((\text{n\_visible},)), W, bv_t, bh_t, k=25)
        u_t = \text{T.tanh}(bu + T.dot(v_t, W_{vu}) + T.dot(u_tm1, W_{uu}))
    else:
        v_t, u_t = \text{build\_rbm}(v, W, bv[:], bh[:], k=15)
        return ([v_t, u_t], updates)

# For training, the deterministic recurrence is used to compute all the
# {bv_t, bh_t, 1 <= t <= T} given v. Conditional RBMs can then be trained
# in batches using those parameters.
(u_t, bv_t, bh_t), updates_train = \text{theano.scan}(
    \text{lambda} v_t, u_tm1, *+: recurrence(v_t, u_tm1),
    sequences=v, outputs_info=[u0, None, None], non_sequences=params)
(v_sample, cost, monitor, updates_rbm = \text{build\_rbm}(v, W, bv[:], bh[:],
    k=15)
updates_train.update(updates_rbm)

# symbolic loop for sequence generation
(v_t, u_t), updates_generate = \text{theano.scan}(
    \text{lambda} u_tm1, *+: recurrence(None, u_tm1),
    outputs_info=[None, u0], non_sequences=params, n_steps=200)

return (v, v_sample, cost, monitor, params, updates_train, v_t, updates_generate)
14.2.3 Putting it all together

We now have all the necessary ingredients to start training our network on real symbolic sequences of polyphonic music.

```python
class RnnRbm:
    '''Simple class to train an RNN-RBM from MIDI files and to generate sample
    sequences.'''
    def __init__(
        self,
        n_hidden=150,
        n_hidden_recurrent=100,
        lr=0.001,
        r=(21, 109),
        dt=0.3
    ):
        '''Constructs and compiles Theano functions for training and sequence
        generation.
        
        n_hidden : integer
            Number of hidden units of the conditional RBMs.
        n_hidden_recurrent : integer
            Number of hidden units of the RNN.
        lr : float
            Learning rate
        r : (integer, integer) tuple
            Specifies the pitch range of the piano-roll in MIDI note numbers,
            including r[0] but not r[1], such that r[1]-r[0] is the number of
            visible units of the RBM at a given time step. The default (21,
            109) corresponds to the full range of piano (88 notes).
        dt : float
            Sampling period when converting the MIDI files into piano-rolls,
            or equivalently the time difference between consecutive time steps.''
        
        self.r = r
        self.dt = dt
        (v, v_sample, cost, monitor, params, updates_train, v_t,
        updates_generate) = build_rnnrbm(
            r[1] - r[0],
            n_hidden,
            n_hidden_recurrent
        )
        gradient = T.grad(cost, params, consider_constant=[v_sample])
        updates_train.update(  
            ((p, p - lr * g) for p, g in zip(params, gradient))
        )
        self.train_function = theano.function(  
            [v],
            monitor,
        )
```

updates=updates_train
}
self.generate_function = theano.function(
[[]
  v_t,
  updates=updates_generate
)

def train(self, files, batch_size=100, num_epochs=200):
  '''Train the RNN-RBM via stochastic gradient descent (SGD) using MIDI
  files converted to piano-rolls.

  files : list of strings
    List of MIDI files that will be loaded as piano-rolls for training.
  batch_size : integer
    Training sequences will be split into subsequences of at most this size
    before applying the SGD updates.
  num_epochs : integer
    Number of epochs (pass over the training set) performed. The user can
    safely interrupt training with Ctrl+C at any time.'''

  assert len(files) > 0, 'Training set is empty!' \
    ' (did you download the data files?)'
  dataset = [midiread(f, self.r, self.dt).piano_roll.astype(theano.config.floatX) \n    for f in files]
  try:
    for epoch in range(num_epochs):
      numpy.random.shuffle(dataset)
      costs = []
      for s, sequence in enumerate(dataset):
        for i in range(0, len(sequence), batch_size):
          cost = self.train_function(sequence[i:i + batch_size])
        costs.append(cost)
      print('Epoch %i/%i' % (epoch + 1, num_epochs))
      print(numpy.mean(costs))
      sys.stdout.flush()
  except KeyboardInterrupt:
    print('Interrupted by user.(')

def generate(self, filename, show=True):
  '''Generate a sample sequence, plot the resulting piano-roll and save it
  as a MIDI file.

  filename : string
    A MIDI file will be created at this location.
  show : boolean
    If True, a piano-roll of the generated sequence will be shown.'''
piano_roll = self.generate_function()
midiwrite(filename, piano_roll, self.r, self.dt)
if show:
    extent = (0, self.dt * len(piano_roll)) + self.r
    pylab.figure()
    pylab.imshow(piano_roll.T, origin='lower', aspect='auto',
                 interpolation='nearest', cmap=pylab.cm.gray_r,
                 extent=extent)
    pylab.xlabel('time (s)')
    pylab.ylabel('MIDI note number')
    pylab.title('generated piano-roll')

14.3 Results

We ran the code on the Nottingham database for 200 epochs; training took approximately 24 hours.
The output was the following:

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/200</td>
<td>-15.0308940028</td>
</tr>
<tr>
<td>2/200</td>
<td>-10.4892606673</td>
</tr>
<tr>
<td>3/200</td>
<td>-10.2394696138</td>
</tr>
<tr>
<td>4/200</td>
<td>-10.1431669994</td>
</tr>
<tr>
<td>5/200</td>
<td>-9.7005382843</td>
</tr>
<tr>
<td>6/200</td>
<td>-8.5985647524</td>
</tr>
<tr>
<td>7/200</td>
<td>-8.35115428534</td>
</tr>
<tr>
<td>8/200</td>
<td>-8.26453580552</td>
</tr>
<tr>
<td>9/200</td>
<td>-8.21208991542</td>
</tr>
<tr>
<td>10/200</td>
<td>-8.16847274143</td>
</tr>
</tbody>
</table>

... truncated for brevity ...

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>190/200</td>
<td>-4.74799179994</td>
</tr>
<tr>
<td>191/200</td>
<td>-4.73488515216</td>
</tr>
<tr>
<td>192/200</td>
<td>-4.7326138489</td>
</tr>
<tr>
<td>193/200</td>
<td>-4.73841636884</td>
</tr>
<tr>
<td>194/200</td>
<td>-4.70255511452</td>
</tr>
<tr>
<td>195/200</td>
<td>-4.71872634914</td>
</tr>
<tr>
<td>196/200</td>
<td>-4.7276415885</td>
</tr>
<tr>
<td>197/200</td>
<td>-4.73497644728</td>
</tr>
<tr>
<td>198/200</td>
<td>-inf</td>
</tr>
<tr>
<td>199/200</td>
<td>-4.75554987143</td>
</tr>
<tr>
<td>200/200</td>
<td>-4.72591935412</td>
</tr>
</tbody>
</table>

The figures below show the piano-rolls of two sample sequences and we provide the corresponding MIDI files:
14.3. Results

Fig. 14.1: Listen to sample1.mid

Fig. 14.2: Listen to sample2.mid
14.4 How to improve this code

The code shown in this tutorial is a stripped-down version that can be improved in the following ways:

- **Preprocessing:** transposing the sequences in a common tonality (e.g. C major / minor) and normalizing the tempo in beats (quarternotes) per minute can have the most effect on the generative quality of the model.

- **Pretraining techniques:** initialize the $W, b_v, b_h$ parameters with independent RBMs with fully shuffled frames (i.e. $W_{uh} = W_{uv} = W_{uu} = W_{vu} = 0$); initialize the $W_{uv}, W_{uu}, W_{vu}, b_u$ parameters of the RNN with the auxiliary cross-entropy objective via either SGD or, preferably, Hessian-free optimization [BoulangerLewandowski12].

- **Optimization techniques:** gradient clipping, Nesterov momentum and the use of NADE for conditional density estimation.

- **Hyperparameter search:** learning rate (separately for the RBM and RNN parts), learning rate schedules, batch size, number of hidden units (recurrent and RBM), momentum coefficient, momentum schedule, Gibbs chain length $k$ and early stopping.

- **Learn the initial condition** $u^{(0)}$ as a model parameter.

A few samples generated with code including these features are available here: sequences.zip.
15.1 Plotting Samples and Filters

Note: The code for this section is available for download here.

To plot a sample, what we need to do is to take the visible units, which are a flattened image (there is no 2D structure to the visible units, just a 1D string of unit activations) and reshape it into a 2D image. The order in which the points from the 1D array go into the 2D image is given by the order in which the initial MNIST images were converted into a 1D array. Lucky for us this is just a call of the numpy.reshape function.

Plotting the weights is a bit more tricky. We have $n_{\text{hidden}}$ hidden units, each of them corresponding to a column of the weight matrix. A column has the same shape as the visible, where the weight corresponding to the connection with visible unit $j$ is at position $j$. Therefore, if we reshape every such column, using numpy.reshape, we get a filter image that tells us how this hidden unit is influenced by the input image.

We need a utility function that takes a minibatch, or the weight matrix, and converts each row (for the weight matrix we do a transpose) into a 2D image and then tile these images together. Once we converted the minibatch or the weights in this image of tiles, we can use PIL to plot and save. PIL is a standard python library to deal with images.

Tiling minibatches together is done for us by the tile_raster_image function which we provide here.

```python
def scale_to_unit_interval(ndar, eps=1e-8):
    """ Scales all values in the ndarray ndar to be between 0 and 1 ""
    ndar = ndar.copy()
    ndar -= ndar.min()
    ndar *= 1.0 / (ndar.max() + eps)
    return ndar
```
def tile_raster_images(X, img_shape, tile_shape, tile_spacing=(0, 0),
                         scale_rows_to_unit_interval=True,
                         output_pixel_vals=True):
    """
    Transform an array with one flattened image per row, into an array in
    which images are reshaped and layed out like tiles on a floor.

    This function is useful for visualizing datasets whose rows are images,
    and also columns of matrices for transforming those rows
    (such as the first layer of a neural net).

    :param X: a 2-D ndarray or a tuple of 4 channels, elements of which can
              be 2-D ndarrays or None;
    :param X: a 2-D array in which every row is a flattened image.

    :type img_shape: tuple; (height, width)
    :type img_shape: the original shape of each image

    :type tile_shape: tuple; (rows, cols)
    :param tile_shape: the number of images to tile (rows, cols)

    :param output_pixel_vals: if output should be pixel values (i.e. int8
                              values) or floats

    :param scale_rows_to_unit_interval: if the values need to be scaled before
                                          being plotted to [0,1] or not

    :returns: array suitable for viewing as an image.
    (See:`Image.fromarray`.)
    :rtype: a 2-d array with same dtype as X.
    """

    assert len(img_shape) == 2
    assert len(tile_shape) == 2
    assert len(tile_spacing) == 2

    # The expression below can be re-written in a more C style as
    # follows :
    #
    # out_shape = [0,0]
    # out_shape[0] = (img_shape[0] + tile_spacing[0]) * tile_shape[0] -
    #    tile_spacing[0]
    # out_shape[1] = (img_shape[1] + tile_spacing[1]) * tile_shape[1] -
    #    tile_spacing[1]
    # out_shape = [(ishp + tsp) * tshp - tsp for ishp, tshp, tsp
    #    in zip(img_shape, tile_shape, tile_spacing)]

    if isinstance(X, tuple):
        assert len(X) == 4
        # Create an output numpy ndarray to store the image
if output_pixel_vals:
    out_array = numpy.zeros((out_shape[0], out_shape[1], 4), dtype='\texttt{uint8}')
else:
    out_array = numpy.zeros((out_shape[0], out_shape[1], 4), dtype=X.dtype)

# colors default to 0, alpha defaults to 1 (opaque)
if output_pixel_vals:
    channel_defaults = [0, 0, 0, 255]
else:
    channel_defaults = [0., 0., 0., 1.]

for i in range(4):
    if X[i] is None:
        # if channel is None, fill it with zeros of the correct dtype
        out_array[:, :, i] = numpy.zeros(out_shape,
                                          dtype='\texttt{uint8}' if output_pixel_vals else out_array.dtype) + channel_defaults[i]
    else:
        # use a recurrent call to compute the channel and store it in the output
        out_array[:, :, i] = tile_raster_images(X[i], img_shape, tile_shape, tile_spacing, scale_rows_to_unit_interval, output_pixel_vals)
    return out_array

else:
    # if we are dealing with only one channel
    H, W = img_shape
    Hs, Ws = tile_spacing

    # generate a matrix to store the output
    out_array = numpy.zeros((H, W, out_shape[0], out_shape[1], 4), dtype='\texttt{uint8}' if output_pixel_vals else X.dtype)

    for tile_row in range(tile_shape[0]):
        for tile_col in range(tile_shape[1]):
            if tile_row * tile_shape[1] + tile_col < X.shape[0]:
                if scale_rows_to_unit_interval:
                    # if we should scale values to be between 0 and 1
                    # do this by calling the `scale_to_unit_interval` function
                    this_img = scale_to_unit_interval(X[tile_row * tile_shape[1] + tile_col].reshape(img_shape))
                else:
                    this_img = X[tile_row * tile_shape[1] + tile_col].reshape(img_shape)
                # add the slice to the corresponding position in the output array
                out_array[
tile_col * (W+Ws): tile_col * (W + Ws) + W
] \n= this_img * (255 if output_pixel_vals else 1)
return out_array
References


[Lee08] 8. Lee, C. Ekanadham, and A.Y. Ng., Sparse deep belief net model for visual area V2, in Ad-

unsupervised learning of hierarchical representations.”, ICML 2009

Modeling Natural Images”. Proc. of the 13-th International Conference on Artificial Intelligence
and Statistics (AISTATS 2010), Italy, 2010

of Sparse Representations with an Energy-Based Model, Advances in Neural Information Processing

nition with
Tomaso.

Features with Denoising Autoencoders, Proceedings of the Twenty-fifth International Conference

[Tieleman08] 20. Tieleman, Training restricted boltzmann machines using approximations to the likeli-
hood gradient, ICML 2008.

[Xavier10] 25. Bengio, X. Glorot, Understanding the difficulty of training deep feedforward neuronal-
networks, AISTATS 2010
Index

D
Dataset notation, 7
Datasets, 5
Download:, 5

E
Early-Stopping, 13

L
L1 and L2 regularization, 12
List of Symbols and acronyms, 8
Logistic Regression, 16

M
Math Convetions, 8
MNIST Dataset, 5
Multilayer Perceptron, 35

N
Negative Log–Likelihood Loss, 9
Notation, 7

P
Python Namespaces, 8

R
Regularization, 12

S
Stochastic Gradient Descent, 10

T
Testing, 14

Z
Zero-One Loss, 9