# Theano Tutorial

[MAREK](http://www.marekrei.com/blog/author/marek/)[JANUARY 25, 2016](http://www.marekrei.com/blog/theano-tutorial/)[UNCATEGORIZED](http://www.marekrei.com/blog/category/uncategorized/)[53 COMMENTS](http://www.marekrei.com/blog/theano-tutorial/" \l "comments)

This is an introductory tutorial on using Theano, the Python library. I’m going to start from scratch and assume no previous knowledge of Theano. However, understanding how neural networks work will be useful when getting to the code examples towards the end.

The plan for the tutorial is as follows:

1. Give a basic introduction to Theano and explain the important concepts.
2. Go over the main operations that we have available in Theano.
3. Look at working code examples.

I recently gave this tutorial as a talk in University of Cambridge and it turned out to be way more popular than expected. In order to give more people access to the material, I’m now writing it up as a blog post.

I do not claim to know everything about Theano, and I constantly learn new things myself. If you find any errors or have suggestions on how to improve this tutorial, do let me know.

The code examples can be found in the Github repository: <https://github.com/marekrei/theano-tutorial>

## 1. What is Theano?



Theano is a Python library for efficiently handling mathematical expressions involving multi-dimensional arrays (also known as tensors). It is a common choice for implementing neural network models. Theano has been developed in University of Montreal, in a group led by Yoshua Bengio, since 2008.

Some of the features include:

* automatic differentiation – you only have to implement the forward (prediction) part of the model, and Theano will automatically figure out how to calculate the gradients at various points, allowing you to perform gradient descent for model training.
* transparent use of a GPU – you can write the same code and run it either on CPU or GPU. More specifically, Theano will figure out which parts of the computation should be moved to the GPU.
* speed and stability optimisations – Theano will internally reorganise and optimise your computations, in order to make them run faster and be more numerically stable. It will also try to compile some operations into C code, in order to speed up the computation.

Technically, Theano isn’t actually a machine learning library, as it doesn’t provide you with pre-built models that you can train on your dataset. Instead, it is a mathematical library that provides you with tools to build your own machine learning models. But if you are looking for machine learning toolkits, there are several good ones implemented on top of Theano:

* Blocks   <http://blocks.readthedocs.org/en/latest/>
* Keras   <http://keras.io/>
* Lasagne   <http://lasagne.readthedocs.org/en/latest/>
* PyLearn2   <http://deeplearning.net/software/pylearn2/>

## 2. Python refresher

Theano is a Python library, so let’s go over some important points in Python.

* Python is an interpreted language, which makes it more platform independent but generally slower than C, for example.
* Python uses dynamic typing. While each variable does have a specific type during execution, these are not explicitly stated in the code.
* Python uses indentation for block delimiting. So where C or Java would use curly brackets to separate a block, Python uses whitespace. Here we define a function f to take parameter x and return 2\*x:

|  |  |
| --- | --- |
| 1  2 | **def** f(x):  **return** 2**\***x |

* We define a list in Python with square brackets:

|  |  |
| --- | --- |
| 1  2 | a **=** [1,2,3,4,5]  a[1] **==** 2 |

* We define a dictionary (key-value mapping) with curly brackets:

|  |  |
| --- | --- |
| 1  2 | b **=** {'key1': 1, 'key2':2}  b['key2'] **==** 2 |

* List comprehension is a neat shorthand in Python for constructing lists. Here we loop for 5 steps (values 0-4), and each time add i+1 to the list:

|  |  |
| --- | --- |
| 1  2 | c **=** [i**+**1 **for** i **in** range(5)]  c[1] **==** 2 |

## 3. Using Theano

In order to use Theano, you will need to install the dependencies and install Theano itself. If you’re using Ubuntu (tested for 14.04), you might get away with just running these two commands:

|  |  |
| --- | --- |
| 1  2 | sudo apt-get install python-numpy python-scipy python-dev python-pip python-nose g++ libopenblas-dev git  sudo pip install Theano |

If that doesn’t work for you, take a look at the original Theano homepage, which contains instructions for various platforms:  
<http://deeplearning.net/software/theano/install.html>

To use Theano in your Python script, include it using:

|  |  |
| --- | --- |
| 1 | **import** theano |

## 4. Minimal Working Example

Here is the smallest example I could come up with, which uses Theano and actually does something:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **import** theano  **import** numpy    x **=** theano.tensor.fvector('x')  W **=** theano.shared(numpy.asarray([0.2, 0.7]), 'W')  y **=** (x **\*** W).sum()    f **=** theano.function([x], y)    output **=** f([1.0, 1.0])  print output |

So what’s happening here?

We first define a Theano variable x to be a vector of 32-bit floats, and give it name ‘x’:

|  |  |
| --- | --- |
| 1 | x **=** theano.tensor.fvector('x') |

Next, we create a Theano variable W, assign its value to be vector [0.2, 0.7], and name it ‘W’:

|  |  |
| --- | --- |
| 1 | W **=** theano.shared(numpy.asarray([0.2, 0.7]), 'W') |

We define y to be the sum of all elements in the element-wise multiplication of x and W:

|  |  |
| --- | --- |
| 1 | y **=** (x **\*** W).sum() |

We define a Theano function f, which takes as input x and outputs y:

|  |  |
| --- | --- |
| 1 | f **=** theano.function([x], y) |

Then call this function, giving as the argument vector [1.0, 1.0], essentially setting the value of variable x:

|  |  |
| --- | --- |
| 1 | output **=** f([1.0, 1.0]) |

The script prints out the summed product of [0.2, 0.7] and [1.0, 1.0], which is:

0.2\*1.0 + 0.7\*1.0 = 0.9

Don’t worry if the code doesn’t fully make sense. We’ll go over the important parts in more detail.

## 5. Symbolic graphs in Theano (!)

I’d say this section contains the most crucial part to understanding Theano.

When we are creating a model with Theano, we first define a symbolic graph of all variables and operations that need to be performed. And then we can apply this graph on specific inputs to get outputs.

For example, what do you think happens when this line of Theano code is executed in our script?

|  |  |
| --- | --- |
| 1 | y **=** (x **\*** W).sum() |

The system takes x and W, multiplies them together and sums the values. Right?

****NOPE****

Instead, we create a Theano object y that knows its values can be calculated as the dot-product of x and W. But the required mathematical operations are not performed here. In fact, when this line was executed in our example code above, x didn’t even have a value yet.

By chaining up various operations, we are creating a graph of all the variables and functions that need to be used to reach the output values. This symbolic graph is also the reason why we can only use Theano-specific operations when defining our models. If we tried to integrate functions from some random Python library into our network, they would attempt to perform the calculations immediately, instead of returning a Theano variable as needed. Exceptions do exist – Theano overrides some basic Python operators to act as expected, and [NumPy](http://www.numpy.org/) is quite well integrated with Theano.

## 6. Variables

We can define variables which don’t have any values yet. Normally, these would be used for inputs to our network.

The variables have to be of a specific type though. For example, here we define variable x to be a vector of 32-bit floats, and give it name ‘x’:

|  |  |
| --- | --- |
| 1 | x **=** theano.tensor.fvector('x') |

The names are generally useful for debugging and informative error messages. Theano won’t have access to your Python variable names, so you have to assign explicit Theano names for each variable if you want them to be referred to as something more useful than just “a tensor”.

There are a number of different variable types available, just have a look at the list [here](http://deeplearning.net/software/theano/library/tensor/basic.html). Some of the more popular ones include:

| **Constructor** | **dtype** | **ndim** |
| --- | --- | --- |
| fvector | float32 | 1 |
| ivector | int32 | 1 |
| fscalar | float32 | 0 |
| fmatrix | float32 | 2 |
| ftensor3 | float32 | 3 |
| dtensor3 | float64 | 3 |

You can also define a generic vector (or tensor) and set the type with an argument:

|  |  |
| --- | --- |
| 1 | x **=** theano.tensor.vector('x', dtype**=**float32) |

If you don’t set the *dtype*, you will create vectors of type config.floatX. This will become relevant in the section about GPUs.

## 7. Shared variables

We can also define shared variables, which are shared between different functions and different function calls. Normally, these would be used for weights in our neural network. Theano will automatically try to move shared variables to the GPU, provided one is available, in order to speed up computation.

Here we define a shared variable and set its value to [0.2, 0.7].

|  |  |
| --- | --- |
| 1 | W **=** theano.shared(numpy.asarray([0.2, 0.7]), 'W') |

The values in shared variables can be accessed and modified outside of our Theano functions using these commands:

|  |  |
| --- | --- |
| 1  2 | W.get\_value()  W.set\_value([0.1, 0.9]) |

## 8. Functions

Theano functions are basically hooks for interacting with the symbolic graph. Commonly, we use them for passing input into our network and collecting the resulting output.

Here we define a Theano function f that takes x as input and returns y as output:

|  |  |
| --- | --- |
| 1 | f **=** theano.function([x], y) |

The first parameter is the list of input variables, and the second parameter is the list of output variables. Although if there’s only one output variable (like now) we don’t need to make it into a list.

When we construct a function, Theano takes over and performs some of its own magic. It builds the computational graph and optimises it as much as possible. It restructures mathematical operations to make them faster and more stable, compiles some parts to C, moves some tensors to the GPU, etc.

Theano compilation can be controlled by setting the value of *mode* in the environement variable THEANO\_FLAGS:

* FAST\_COMPILE – Fast to compile, slow to run. Python implementations only, minimal graph optimisation.
* FAST\_RUN – Slow to compile, fast to run. C implementations where available, full range of optimisations

## 9. Minimal Training Example

Here’s a minimal script for actually training something in Theano. We will be training the weights in W using gradient descent, so that the result from the model would be *20* instead of the original *0.9*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **import** theano  **import** numpy    x **=** theano.tensor.fvector('x')  target **=** theano.tensor.fscalar('target')    W **=** theano.shared(numpy.asarray([0.2, 0.7]), 'W')  y **=** (x **\*** W).sum()    cost **=** theano.tensor.sqr(target **-** y)  gradients **=** theano.tensor.grad(cost, [W])  W\_updated **=** W **-** (0.1 **\*** gradients[0])  updates **=** [(W, W\_updated)]    f **=** theano.function([x, target], y, updates**=**updates)    **for** i **in** xrange(10):      output **=** f([1.0, 1.0], 20.0)      print output |

We create a second input variable called *target*, which will act as the target value we use for training:

|  |  |
| --- | --- |
| 1 | target **=** theano.tensor.fscalar('target') |

In order to train the model, we need a cost function. Here we use a simple squared distance from the target:

|  |  |
| --- | --- |
| 1 | cost **=** theano.tensor.sqr(target **-** y) |

Next, we want to calculate the partial gradients for the parameters that will be updated, with respect to the cost function. Luckily, Theano will do that for us. We simply call the *grad* function, pass in the real-valued cost and a list of all the variables we want gradients for, and it will return a list of those gradients:

|  |  |
| --- | --- |
| 1 | gradients **=** theano.tensor.grad(cost, [W]) |

Now let’s define a symbolic variable for what the updated version of the parameters will look like. Using gradient descent, the update rule is to subtract the gradient, multiplied by the learning rate:

|  |  |
| --- | --- |
| 1 | W\_updated **=** W **-** (0.1 **\*** gradients[0]) |

And next we create a list of updates. More specifically, a list of tuples where the first element is the variable we want to update, and the second element is a variable containing the values that we want the first variable to contain after the update. This is just a syntax that Theano requires.

|  |  |
| --- | --- |
| 1 | updates **=** [(W, W\_updated)] |

Have to define a Theano function again, with a couple of changes:

|  |  |
| --- | --- |
| 1 | f **=** theano.function([x, target], y, updates**=**updates) |

It now takes two input arguments – one for the input vector, and another for the target value used for training. And the list of updates also gets attached to the function as well. Every time this function is called, we pass in values for *x* and *target*, get back the value for *y* as output, and Theano performs all the updates in the update list.

In order to train the parameters, we repeatedly call this function (10 times in this case). Normally, we’d pass in different examples from our training data, but for this example we use the same x=[1.0, 1.0] and target=20 each time:

|  |  |
| --- | --- |
| 1  2  3 | **for** i **in** xrange(10):      output **=** f([1.0, 1.0], 20.0)      print output |

When the script is executed, the output looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 0.9  8.54  13.124  15.8744  17.52464  18.514784  19.1088704  19.46532224  19.679193344  19.8075160064 |

The first time the function is called, the output value is still 0.9 (like in the previous example), because the updates have not been applied yet. But with each consecutive step, the output value becomes closer and closer to the desired target 20.

## 10. Useful operations

This covers the basic logic behind building models with Theano. The example was very simple, but we are free to define increasingly complicated networks, as long as we use Theano-specific functions. Now let’s look at some of these building blocks that we have available.

****Evaluate the value of a Theano variable****

The eval() function forces the Theano variable to calculate and return its actual (numerical) value. If we try to just print the variable *a*, we only print its name. But if we use eval(), we get the actual square matrix that it is initialised to.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | > a **=** theano.shared(numpy.asarray([[1.0,2.0],[3.0,4.0]]), 'a')  > a  a  > a.eval()  array([[1., 2.],         [3., 4.]]) |

This eval() function isn’t really used for building models, but it can be useful for debugging and learning how Theano works. In the examples below, I will be using the matrix *a* and the eval() function to print the value of each variable and demonstrate how different operations work.

****Basic element-wise operations: + – \* /****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** ((a **+** a) **/** 4.0)    array([[ 0.5, 1. ],         [ 1.5, 2. ]]) |

****Dot product****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** theano.tensor.dot(a, a)    array([[ 7., 10.],         [15., 22.]]) |

****Activation functions****

|  |  |
| --- | --- |
| 1  2  3  4  5 | c **=** theano.tensor.nnet.sigmoid(a)  c **=** theano.tensor.tanh(a)    array([[ 0.76159416,  0.96402758],         [ 0.99505475,  0.9993293 ]]) |

****Softmax (row-wise)****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** theano.tensor.nnet.softmax(a)    array([[ 0.26894142,  0.73105858],         [ 0.26894142,  0.73105858]]) |

****Sum****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** a.sum()  c **=** a.sum(axis**=**1)    array([ 3.,  7.]) |

****Max****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** a.max()  c **=** a.max(axis**=**1)    array([ 2.,  4.]) |

****Argmax****

|  |  |
| --- | --- |
| 1  2  3  4 | c **=** theano.tensor.argmax(a)  c **=** theano.tensor.argmax(a, axis**=**1)    array([1, 1]) |

****Reshape****

We sometimes need to change the dimensions of a tensor and reshape() allows us to do that. It takes as input a tuple containing the new shape and returns a new tensor with that shape. In the first example below, we shape a square matrix into a 1×4 matrix. In the second example, we use -1 which means “as big as the dimension needs to be”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a **=** theano.shared(numpy.asarray([[1,2],[3,4]]), 'a')  c **=** a.reshape((1,4))  array([[1, 2, 3, 4]])    c **=** a.reshape((**-**1,))  array([1, 2, 3, 4]) |

****Zeros-like, ones-like****

These functions create new tensors with the same shape but all values set to zero or one.

|  |  |
| --- | --- |
| 1  2  3 | c **=** theano.tensor.zeros\_like(a)  array([[0, 0],         [0, 0]]) |

****Reorder the tensor dimensions****

Sometimes we need to reorder the dimensions in a tensor. In the examples below, the dimensions in a two-dimensional matrix are first swapped. Then, ‘x’ is used to create a brand new dimension.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | a.eval()  array([[1, 2],         [3, 4]])    c **=** a.dimshuffle((1,0))  array([[1, 3],         [2, 4]])    c **=** a.dimshuffle(('x',0,1))  array([[[1, 2],          [3, 4]]]) |

****Indexing****

Using Python indexing tricks can make life so much easier. In the example below, we make a separate list *b* containing line numbers, and use it to construct a new matrix which contains exactly the lines we want from the original matrix. This can be useful when dealing with word embeddings – we can put word ids into a list and use this to retrieve exactly the correct sequence of embeddings from the whole embedding matrix.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | a **=** theano.shared(numpy.asarray([[1.0,2.0],[3.0,4.0]]), 'a')  array([[1., 2.],         [3., 4.]])    b **=** [1,1,0]  c **=** a[b]  array([[ 3.,  4.],         [ 3.,  4.],         [ 1.,  2.]]) |

For assignment, we can’t do this:

|  |  |
| --- | --- |
| 1 | a[0] **=** [0.0, 0.0] |

But instead, we can use set\_subtensor(), which takes as arguments the selection of the original matrix that we want to reassign, and the value we want to assign it to. It returns a new tensor that has the corresponding values modified.

|  |  |
| --- | --- |
| 1  2  3 | c **=** theano.tensor.set\_subtensor(a[0],[0.0, 0.0])  array([[ 0.,  0.],         [ 3.,  4.]]) |

## 11. Classifier Code Example

At this point, it’s time to move on to some more realistic examples.

Take a look at the [code for a very basic classifier](https://github.com/marekrei/theano-tutorial/blob/master/classifier.py), which tries to train a small network on a tiny (but real) dataset. I won’t walk you through it line-by-line any more; you’ve learned all the necessary parts by now and there are comments in the code as well.

The task is to predict whether the GDP per capita for a country is more than the average GDP, based on the following features:

* Population density (per suqare km)
* Population growth rate (%)
* Urban population (%)
* Life expectancy at birth (years)
* Fertility rate (births per woman)
* Infant mortality (deaths per 1000 births)
* Enrolment in tertiary education (%)
* Unemployment (%)
* Estimated control of corruption (score)
* Estimated government effectiveness (score)
* Internet users (per 100 people)

The *data/* directory contains the files for training (121 countries) and testing (40 countries). Each row represents one country, the first column is the label, followed by the features. The feature values have been normalised, by subtracting the mean and dividing by the standard deviation. The label is 1 if the GDP is more than average, and 0 otherwise.

Once you clone the github repository (or just download the data files), you can run the script with:

|  |  |
| --- | --- |
| 1 | python classifier.py data/countries-classify-gdp-normalised.train.txt data/countries-classify-gdp-normalised.test.txt |

The script will print information about 10 training epochs and the result on the test set:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Epoch: 0, Training\_cost: 28.4304042768, Training\_accuracy: 0.578512396694  Epoch: 1, Training\_cost: 24.5186290354, Training\_accuracy: 0.619834710744  Epoch: 2, Training\_cost: 22.1283727037, Training\_accuracy: 0.619834710744  Epoch: 3, Training\_cost: 20.7941253329, Training\_accuracy: 0.619834710744  Epoch: 4, Training\_cost: 19.9641569475, Training\_accuracy: 0.619834710744  Epoch: 5, Training\_cost: 19.3749411377, Training\_accuracy: 0.619834710744  Epoch: 6, Training\_cost: 18.8899216914, Training\_accuracy: 0.619834710744  Epoch: 7, Training\_cost: 18.4006371608, Training\_accuracy: 0.677685950413  Epoch: 8, Training\_cost: 17.7210185975, Training\_accuracy: 0.793388429752  Epoch: 9, Training\_cost: 16.315597037, Training\_accuracy: 0.876033057851  Test\_cost: 5.01800578051, Test\_accuracy: 0.925 |

## 12. Recurrent functions with scan

One more important operation to cover is scan, which can be used to create various recurrent functions: RNN, GRU, LSTM, etc.

Here is sample code for using scan to define a simple RNN over word vectors in the input\_vectors matrix:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **def** rnn\_step(x, previous\_hidden\_vector, W\_input, W\_recurrent):      hidden\_vector **=** theano.tensor.dot(x, W\_input) **+**                      theano.tensor.dot(previous\_hidden\_vector, W\_recurrent)      hidden\_vector **=** theano.tensor.nnet.sigmoid(hidden\_vector)    W\_input **=** self.create\_parameter\_matrix('W\_input', (word\_embedding\_size, recurrent\_size))  W\_recurrent **=** self.create\_parameter\_matrix('W\_recurrent', (recurrent\_size, recurrent\_size))  initial\_hidden\_vector **=** theano.tensor.alloc(numpy.array(0, dtype**=**floatX), recurrent\_size)    hidden\_vector, \_ **=** theano.scan(      rnn\_step,      sequences **=** input\_vectors,      outputs\_info **=** initial\_hidden\_vector,      non\_sequences **=** [W\_input, W\_recurrent]  )    hidden\_vector **=** hidden\_vector[**-**1] |

The scan function is called on line 10 and it takes 4 important arguments:

* fn: The function that is called at every step of the iteration.
* sequences: The variables that we want to iterate over. If this is a matrix, we will be iterating over each row of that matrix.
* outputs\_info: The values that we use as the previous recurrent values for the very first step. Usually these are just set to 0.
* non\_sequences: Any additional variables that we want to pass into the function (fn) but don’t want to iterate over.

We’ve defined the helper function rnn\_step on line 1, which gets called on each row of our input matrix. The scan function will be calling this rnn\_step function internally, so we need to accept any arguments in the same order as Theano passes them. This is just something you need to know when dealing with scan. The order is as follows:

1. First, the current items from the variables that we are iterating over. If we are iterating over a matrix, the current row is passed to the function.
2. Next, anything that was output from the function at the previous time step. This is what we use to build recursive and recurrent representations. At the very first time step, the values will be those from outputs\_info instead.
3. Finally, anything we specified in non\_sequences.

What comes out from the scan function contains the hidden states (eg the rnn\_step outputs) at each step. Not just the last step, but all of them. So if you only want the last step, you need to explicitly retrieve it by indexing from -1 (the last element). Theano is actually smart enough to figure out that you’re only using the last result, and will optimise to discard all the intermediate ones.

In order to construct the weight matrices, I’m using a helper function (self.create\_parameter\_matrix, definition not shown here) which takes as input the variable name and the shape. This means I don’t need to define the weight initialisation part again each time.

## 13. RNN Classifier Code Example

Time to look at some more code, this time using recurrent functions and scan. The [script is available at the Github repository](https://github.com/marekrei/theano-tutorial/blob/master/rnnclassifier.py). In this example, I’m using Gated Recurrent Units (GRU) from “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation” ([Cho et al, 2014](http://arxiv.org/abs/1406.1078)), which are essentially a simpler versions of LSTMs.

The task is to classify sentences into 5 classes, based on their fine-grained sentiment (very negative, slightly negative, neutral, slightly positive, very positive). We use the dataset published in “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank” ([Socher et al., 2013](http://www.aclweb.org/anthology/D13-1170)).

Start by downloading the dataset from <http://nlp.stanford.edu/sentiment/> (the main zip file) and unpack it somewhere. Then, create training and test splits in the format that is more suitable for us, using the provided script in [the repository](https://github.com/marekrei/theano-tutorial):

|  |  |
| --- | --- |
| 1  2 | python stanford\_sentiment\_extractor.py 1 full /path/to/sentiment/dataset/ > data/sentiment.train.txt  python stanford\_sentiment\_extractor.py 2 full /path/to/sentiment/dataset/ > data/sentiment.test.txt |

Now we can run the classifier with:

|  |  |
| --- | --- |
| 1 | python rnnclassifier.py data/sentiment.train.txt data/sentiment.test.txt |

The script will train for 3 passes over the training data, and will then print performance on the test data.

|  |  |
| --- | --- |
| 1  2  3  4 | Epoch: 0 Cost: 25937.7372292 Accuracy: 0.285814606742  Epoch: 1 Cost: 21656.820174 Accuracy: 0.350655430712  Epoch: 2 Cost: 18020.619533 Accuracy: 0.429073033708  Test\_cost: 4784.25137484 Test\_accuracy: 0.388235294118 |

The accuracy on the test set is about 38%, which isn’t a great result. But it is quite a difficult task – the current state-of-the-art system ([Tai ei al., 2015](https://aclweb.org/anthology/P/P15/P15-1150.pdf)) achieves 50.9% accuracy, using a large amount of additional phrase-level annotations, and a much bigger network based on LSTMs and parse trees. As there are 5 classes to choose from, a random system would get 20% accuracy.

## 14. Running on a GPU

Theano is smart enough to move some parts of the processing to the GPU, as long as CUDA is installed and a graphics card is made available. To install CUDA, follow instructions on one of these links:

<https://developer.nvidia.com/cuda-downloads>  
<http://www.r-tutor.com/gpu-computing/cuda-installation/cuda7.5-ubuntu>

Then, when running your Python script, you need to point Theano to the CUDA installation. I do this by setting the environment variables in the command line:

|  |  |
| --- | --- |
| 1 | LD\_LIBRARY\_PATH=/usr/lib:/usr/local/cuda-7.5/lib64 THEANO\_FLAGS='cuda.root=/usr/local/cuda-7.5,device=gpu,floatX=float32' python mycode.py |

This command is for CUDA-7.5 in my system. You’ll need to make sure that the paths match the CUDA installation paths in your machine. If it works and Theano is using a GPU, the first line that gets printed will explicitly say so. Something like this:

|  |  |
| --- | --- |
| 1 | Using gpu device 0: GeForce GTX 780 |

If you don’t get something similar, it probably means Theano is not properly hooked up to use the GPU.

At the time of writing, Theano only supports 32-bit variables on the GPU, and this is where the floatX=float32 setting comes it. It just allows you to set the data type during the execution of the script, without writing it into your code. For example, you can define your vectors like this:

|  |  |
| --- | --- |
| 1 | x **=** theano.tensor.vector('x', dtype**=**config.floatX) |

And now you can set floatX to be float32 when running the script on a GPU and float64 when running on your CPU.

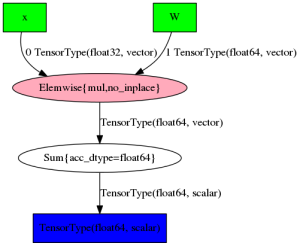
Finally, if your machine has multiple GPUs, you can control which one is used for the script by setting device=gpu0, device=gpu1, etc. Based on personal experience, running multiple Theano jobs on the same GPU does not give any advantage, so it’s best to send them to different ones when possible.

## 15. Drawing the computation graph

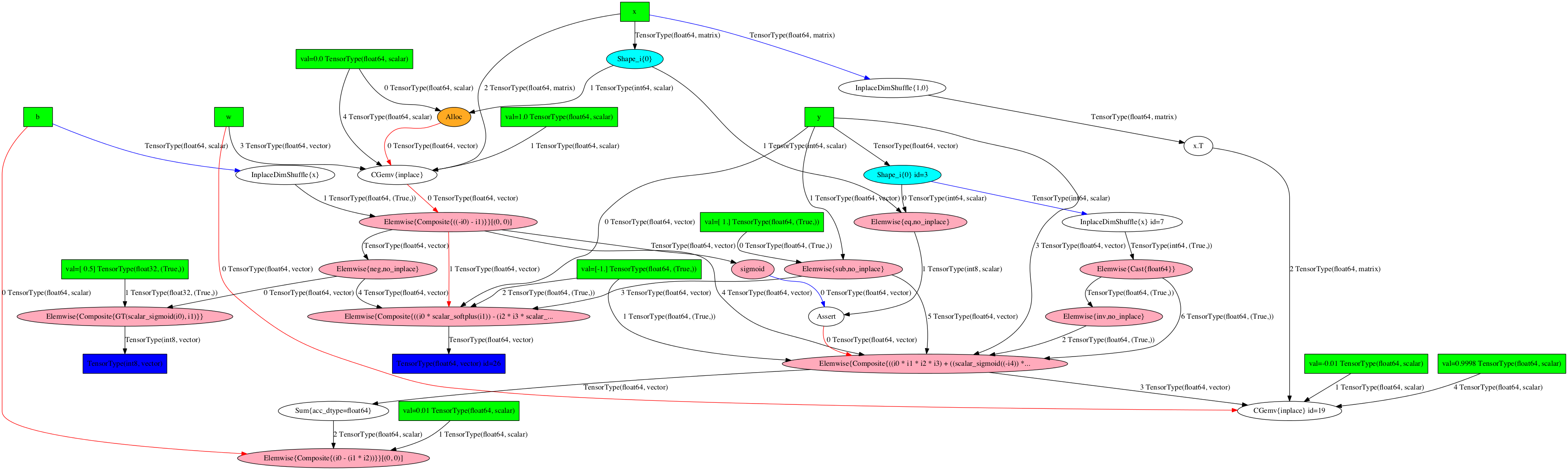
Theano provides a command for printing a variable or a function, along with all the required computation, as an image:

|  |  |
| --- | --- |
| 1  2 | f **=** theano.function([x], y)  theano.printing.pydotprint(f, outfile**=**"f.png", var\_with\_name\_simple**=**True) |

When dealing with very simple models, this can give a nice graphical representation. For example, here is a model from our minimal working example:



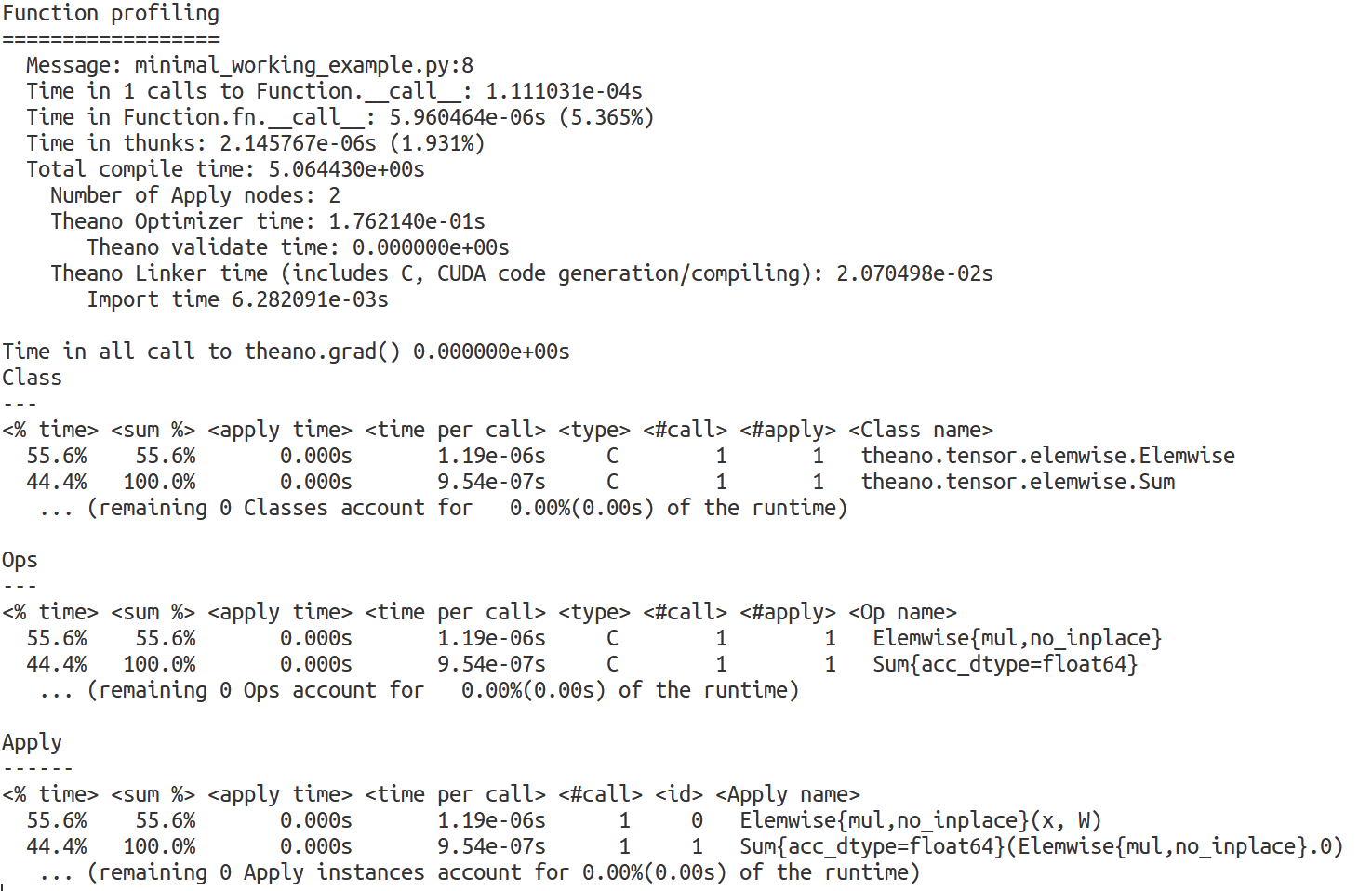
However, when the models get more and more complicated, the images also tend to get less informative:



## 16. Profiling

Finally, Theano also provides a useful tool for analysing bottlenecks in your code. Just set profile=True in THEANO\_FLAGS, and it will print information about how much time is spent on different operations in your code.

|  |  |
| --- | --- |
| 1 | THEANO\_FLAGS**=**'profile=True' python minimal\_working\_example.py |



## 17. References

This concludes the Theano tutorial. If you haven’t yet had enough, take a look at the following links that I used for inspiration:

<https://www.tutorialspoint.com/theano/index.htm>

Theano is a Python library for fast numerical computation that can be run on the CPU or GPU.

It is a key foundational library for Deep Learning in Python that you can use directly to create Deep Learning models or wrapper libraries that greatly simplify the process.

In this post you will discover the Theano Python library.

****Kick-start your project**** with my new book [Deep Learning With Python](https://machinelearningmastery.com/deep-learning-with-python/), including step-by-step tutorials and the Python source code files for all examples.

Let’s get started.



Introduction to the Python Deep Learning Library Theano  
Photo by [Kristoffer Trolle](https://www.flickr.com/photos/kristoffer-trolle/17088729869/), some rights reserved.

## **What is Theano?**

Theano is an open source project released under the BSD license and was developed by the LISA (now [MILA](http://mila.umontreal.ca/)) group at the University of Montreal, Quebec, Canada (home of [Yoshua Bengio](http://www.iro.umontreal.ca/~bengioy/yoshua_en/index.html)). It is named after a [Greek mathematician](https://en.wikipedia.org/wiki/Theano_(philosopher)).

At it’s heart Theano is a compiler for mathematical expressions in Python. It knows how to take your structures and turn them into very efficient code that uses NumPy, efficient native libraries like [BLAS](http://www.netlib.org/blas/) and native code (C++) to run as fast as possible on CPUs or GPUs.

It uses a host of clever code optimizations to squeeze as much performance as possible from your hardware. If you are into the nitty-gritty of mathematical optimizations in code, [check out this interesting list](http://deeplearning.net/software/theano/optimizations.html" \l "optimizations).

The actual syntax of Theano expressions is symbolic, which can be off putting to beginners used to normal software development. Specifically, expression are defined in the abstract sense, compiled and later actually used to make calculations.

It was specifically designed to handle the types of computation required for large neural network algorithms used in Deep Learning. It was one of the first libraries of its kind (development started in 2007) and is considered an industry standard for Deep Learning research and development.

## **How to Install Theano**

Theano provides extensive installation instructions for the major operating systems: Windows, OS X and Linux. Read the [Installing Theano guide for your platform](http://deeplearning.net/software/theano/install.html).

Theano assumes a working Python 2 or Python 3 environment with [SciPy](https://www.scipy.org/). There are ways to make the installation easier, such as using [Anaconda](https://www.continuum.io/downloads) to quickly set up Python and SciPy on your machine as well as using [Docker images](http://deeplearning.net/software/theano/install.html" \l "docker-images).

With a working Python and SciPy environment, it is relatively straightforward to install Theano. from PyPI using pip, for example:

|  |  |
| --- | --- |
| 1 | pip install Theano |

At the time of writing the last official release of Theano was version 0.8 which was released 21th March 2016.

New releases may be announced and you will want to update to get any bug fixes and efficiency improvements. You can upgrade Theano using pip as follows:

|  |  |
| --- | --- |
| 1 | sudo pip install --upgrade --no-deps theano |

You may want to use the bleeding edge version of Theano checked directly out of Github.

This may be required for some wrapper libraries that make use of bleeding edge API changes. You can install Theano directly from a Github checkout as follows:

|  |  |
| --- | --- |
| 1 | pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git |

You are now ready to run Theano on your CPU, which is just fine for the development of small models.

Large models may run slowly on the CPU. If you have a Nvidia GPU, you may want to look into configuring Theano to use your GPU. Read the [Using the GPU guides for Linux](http://deeplearning.net/software/theano/install.html" \l "using-the-gpu) or [Mac OS X to set up Theano to use the GPU](http://deeplearning.net/software/theano/install.html" \l "gpu-macos) and the [Using the GPU guide](http://deeplearning.net/software/theano/tutorial/using_gpu.html) for how to test whether it is working.

## **Simple Theano Example**

In this section we demonstrate a simple Python script that gives you a flavor of Theano.

It is taken from the [Theano at a Glance guide](http://deeplearning.net/software/theano/introduction.html). In this example we define two symbolic floating point variables a and b.

We define an expression that uses these variables (c = a + b).

We then compile this symbolic expression into a function using Theano that we can use later.

Finally, we use our complied expression by plugging in some real values and performing the calculation using efficient compiled Theano code under the covers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import theano  from theano import tensor  # declare two symbolic floating-point scalars  a = tensor.dscalar()  b = tensor.dscalar()  # create a simple expression  c = a + b  # convert the expression into a callable object that takes (a,b)  # values as input and computes a value for c  f = theano.function([a,b], c)  # bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'  assert 4.0 == f(1.5, 2.5) |

Running the example does not provide any output. The assertion that 1.5 + 2.5 = 4.0 is true.

This is a useful example as it gives you a flavor for how a symbolic expression can be defined, compiled and used. You can see how this may be scaled up to large vector and matrix operations required for deep learning.

## **Extensions and Wrappers for Theano**

If you are new to deep learning you do not have to use Theano directly.

In fact, you are highly encouraged to use one of many popular Python projects that make Theano a lot easier to use for deep learning.

These projects provide data structures and behaviors in Python, specifically designed to quickly and reliably create deep learning models whilst ensuring that fast and efficient models are created and executed by Theano under the covers.

The amount of Theano syntax exposed by the libraries varies.

* For example the [Lasagne library](https://lasagne.readthedocs.org/en/latest/) provides convenience classes for creating deep learning model but still expects you to know and make use of Theano syntax. This is good for beginners that know or are willing to learn a little Theano as well.
* Another example is [Keras](http://keras.io/) that hides Theano completely and provides a very simple API to work with to create Deep Learning models. It hides Theano so well, that it can in fact run as a wrapper for another popular foundation framework called [TensorFlow](https://www.tensorflow.org/).

I highly recommend experimenting a little with Theano directly, then selecting a wrapper library to learn and practice deep learning.

For a full list of libraries built on Theano see the [Related projects guide](https://github.com/Theano/Theano/wiki/Related-projects) on the Theano Wiki.

https://theano-pymc.readthedocs.io/en/latest/tutorial/index.html