Theano

A short practical guide

Emmanuel Bengio
folinoid.com

What is Theano?

- A language
- A compiler
- A Python library

```python
import theano
import theano.tensor as T
```

What is Theano?

What you really do:

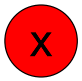
- Build **symbolic** graphs of computation (w/ input nodes)
- Automatically compute gradients through it

```
gradient = T.grad(cost, parameter)
```

- Feed some data
- Get results!

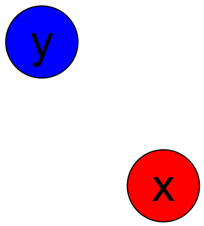
First Example

```
x = T.scalar('x')
```



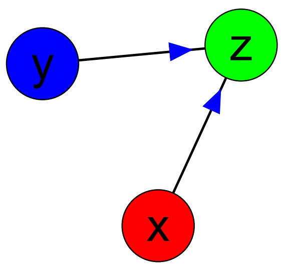
First Example

```
x = T.scalar('x')
y = T.scalar('y')
```



First Example

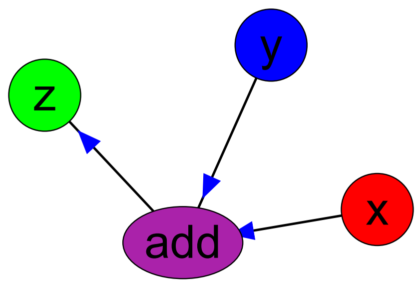
```
x = T.scalar('x')
y = T.scalar('y')
z = x + y
```



First Example

```
x = T.scalar('x')
y = T.scalar('y')
z = x + y
```

`'add'` is an **Op**.



Ops in 1 slide

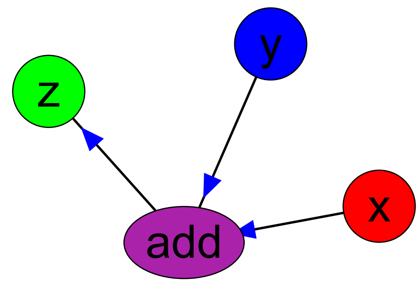
Ops are the building blocks of the computation graph

They (usually) define:

- A computation (given inputs)
- A partial gradient (given inputs and output gradients)
- C/CUDA code that does the computation

First Example

```
x = T.scalar()
y = T.scalar()
z = x + y
f = theano.function([x,y],z)
f(2,8) # 10
```

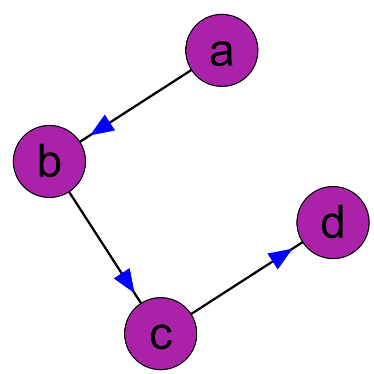


A 5 line Neural Network (evaluator)

```
x = T.vector('x')
W = T.matrix('weights')
b = T.vector('bias')
z = T.nnet.softmax(T.dot(x,W) + b)
f = theano.function([x,W,b],z)
```

A parenthesis about The Graph

```
a = T.vector()
b = f(a)
c = g(b)
d = h(c)
full_fun = theano.function([a],d) # h(g(f(a)))
part_fun = theano.function([c],d) # h(c)
```



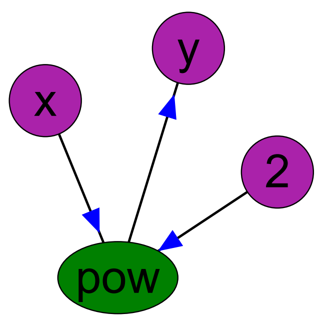
Remember the chain rule?

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial z}$$

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial c}\cdots\frac{\partial x}{\partial y}\frac{\partial y}{\partial z}$$

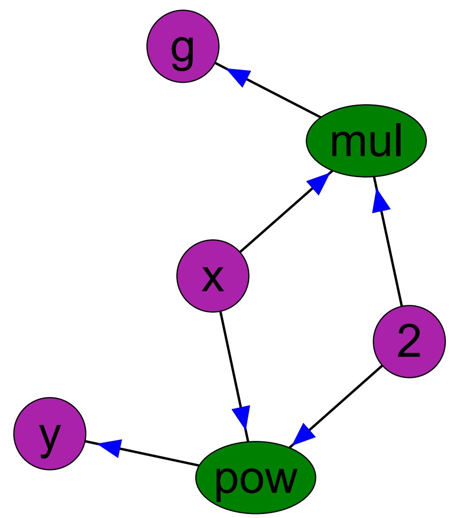
T.grad

```
x = T.scalar()
y = x ** 2
```



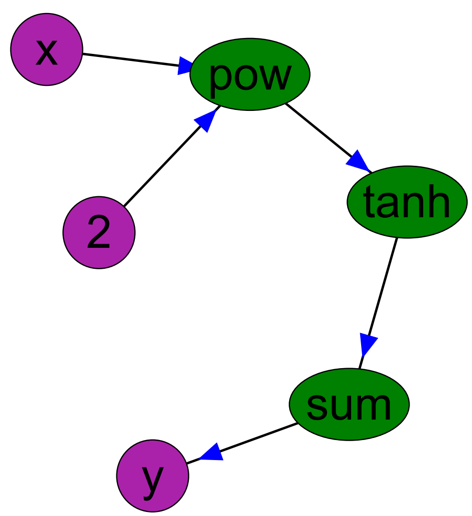
T.grad

```
x = T.scalar()
y = x ** 2
g = T.grad(y, x) # 2*x
```



T.grad

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial c}\cdots\frac{\partial x}{\partial y}\frac{\partial y}{\partial z}$$



`T.grad` take home

You don't really need to think about the gradient anymore.

- all you need is a **scalar** cost
- some parameters
- and a call to `T.grad`

Shared variables

(or, wow, sending things to the GPU is long)

Data reuse is made through 'shared' variables.

```
initial_W = uniform(-k,k,(n_in, n_out))
W = theano.shared(value=initial_W, name="W")
```

That way it sits in the 'right' memory spots

(e.g. on the GPU if that's where your computation happens)

Shared variables

Shared variables act like any other node:

```
prediction = T.dot(x,W) + b
cost = T.sum((prediction - target)**2)
gradient = T.grad(cost, W)
```

You can compute stuff, take gradients.

Shared variables : updating

Most importantly, you can:

update their value, during a function call:

```
gradient = T.grad(cost, W)
update_list = [(W, W - lr * gradient)]
f = theano.function(
       [x,y,lr],[cost],
       updates=update_list)
```

Remember, `theano.function` only builds a function.

```
# this updates W
f(minibatch_x, minibatch_y, learning_rate)
```

Shared variables : dataset

If dataset is small enough, use a shared variable

```
index = T.iscalar()
X = theano.shared(data['X'])
Y = theano.shared(data['Y'])
f = theano.function(
       [index,lr],[cost],
       updates=update_list,
       givens={x:X[index], y:Y[index]})
```

You can also take slices: `X[idx:idx+n]`

Printing things

There are 3 major ways of printing values:

1. When building the graph
2. During execution
3. After execution

And you should do a lot of 1 and 3

Printing things when building the graph

Use a test value

```
# activate the testing
theano.config.compute_test_value = 'raise'
x = T.matrix()
x.tag.test_value = numpy.ones((mbs, n_in))
y = T.vector()
y.tag.test_value = numpy.ones((mbs,))
```

You should do this when designing your model to:

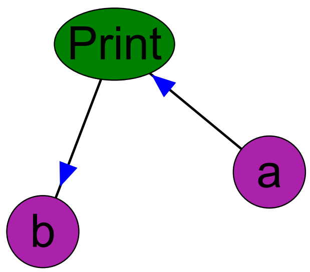
- test shapes
- test types
- ...

Now every node has a `.tag.test_value`

Printing things when executing a function

Use the `Print` Op.

```
from theano.printing import Print
a = T.nnet.sigmoid(h)
# this prints "a:", a.__str__ and a.shape
a = Print("a",["__str__","shape"])(a)
b = something(a)
```



- `Print` acts like the identity
- gets activated whenever b "requests" a
- anything in `dir(numpy.ndarray)` goes

Printing things after execution

Add the node to the outputs

```
theano.function([...],
                [..., some_node])
```

Any node can be an output (even inputs!)

You should do this:

- To acquire statistics
- To monitor gradients, activations...
- With moderation*

*especially on GPU, as this sends all the data back to the CPU at each call

Shapes, dimensions, and shuffling

You can reshape arrays:

```
b = a.reshape((n,m,p))
```

As long as their *flat* dimension is $n \times m \times p$

Shapes, dimensions, and shuffling

You can change the dimension order:

```
# b[i,k,j] == a[i,j,k]
b = a.dimshuffle(0,2,1)
```

Shapes, dimensions, and shuffling

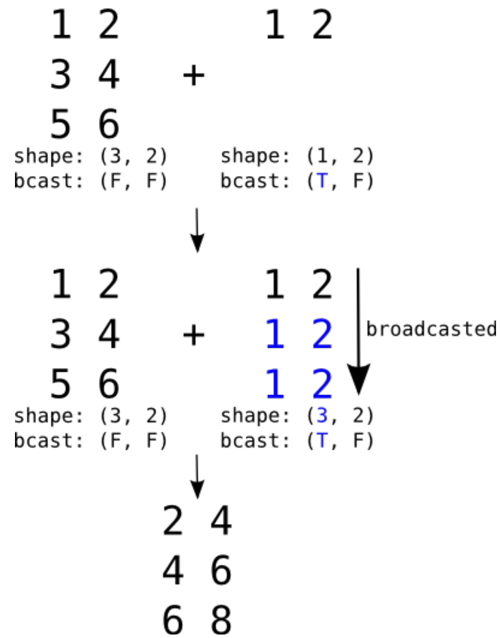
You can also add **broadcast dimensions**:

```
# a.shape == (n,m)
b = a.dimshuffle(0,'x',1)
# or
b = a.reshape([n,1,m])
```

This allows you to do elemwise* operations with b as if it was $n \times p \times m$, where p can be arbitrary.

* e.g. addition, multiplication

Broadcasting



If an array lacks dimensions to match the other operand, the broadcast pattern is automatically expended

to the **left** ((F,) $\longrightarrow$ (T, F), $\longrightarrow$ (T, T, F), ...),

to match the number of dimensions

(But you should always do it yourself)

Profiling

When compiling a function, ask theano to profile it:

```
f = theano.function(..., profile=True)
```

when exiting python, it will print the profile.

Profiling

```
Class
---
<% time> < sum %>< apply time>< time per call>< type><#call>  <#apply> < Class name>
  30.4%    30.4%     10.202s       5.03e-05s     C   202712       4   theano.sandbox.cuda.basic_ops.GpuFromHost
  23.8%    54.2%      7.975s       1.31e-05s     C   608136      12   theano.sandbox.cuda.basic_ops.GpuElemwise
  18.3%    72.5%      6.121s       3.02e-05s     C   202712       4   theano.sandbox.cuda.blas.GpuGemv
   6.0%    78.5%      2.021s       1.99e-05s     C   101356       2   theano.sandbox.cuda.blas.GpuGer
   4.1%    82.6%      1.368s       2.70e-05s     Py   50678       1   theano.tensor.raw_random.RandomFunction
   3.5%    86.1%      1.172s       1.16e-05s     C   101356       2   theano.sandbox.cuda.basic_ops.HostFromGpu
   3.1%    89.1%      1.027s       2.03e-05s     C    50678       1   theano.sandbox.cuda.dnn.GpuDnnSoftmaxGrad
   3.0%    92.2%      1.019s       2.01e-05s     C    50678       1   theano.sandbox.cuda.nnet.GpuSoftmaxWithBias
   2.8%    94.9%      0.938s       1.85e-05s     C    50678       1   theano.sandbox.cuda.basic_ops.GpuCAReduce
   2.4%    97.4%      0.810s       7.99e-06s     C   101356       2   theano.sandbox.cuda.basic_ops.GpuAllocEmpty
   0.8%    98.1%      0.256s       4.21e-07s     C   608136      12   theano.sandbox.cuda.basic_ops.GpuDimShuffle
   0.5%    98.6%      0.161s       3.18e-06s     Py   50678       1   theano.sandbox.cuda.basic_ops.GpuFlatten
   0.5%    99.1%      0.156s       1.03e-06s     C   152034       3   theano.sandbox.cuda.basic_ops.GpuReshape
   0.2%    99.3%      0.075s       4.94e-07s     C   152034       3   theano.tensor.elemwise.Elemwise
   0.2%    99.5%      0.073s       4.83e-07s     C   152034       3   theano.compile.ops.Shape_i
   0.2%    99.7%      0.070s       6.87e-07s     C   101356       2   theano.tensor.opt.MakeVector
   0.1%    99.9%      0.048s       4.72e-07s     C   101356       2   theano.sandbox.cuda.basic_ops.GpuSubtensor
   0.1%   100.0%      0.029s       5.80e-07s     C    50678       1   theano.tensor.basic.Reshape
   0.0%   100.0%      0.015s       1.47e-07s     C   101356       2   theano.sandbox.cuda.basic_ops.GpuContiguous
   ... (remaining 0 Classes account for   0.00%(0.00s) of the runtime)
```

Finding the culprits:

```
24.1% 24.1% 4.537s 1.59e-04s 28611 2 GpuFromHost(x)
```

Profiling

A few common names:

- **Gemm/Gemv**, matrix$\times$matrix / matrix$\times$vector
- **Ger**, matrix update
- **GpuFromHost**, data CPU $\rightarrow$ GPU
- **HostFromGPU**, the opposite
- **[Advanced]Subtensor**, indexing
- **Elemwise**, element-per-element Ops (+, -, exp, log, ...)
- **Composite**, many elemwise Ops merged together.

Loops and recurrent models

Theano has loops, but can be quite complicated.

So here's a simple example

```
x = T.vector('x')
n = T.scalar('n')
def inside_loop(x_t, acc, n):
  return acc + x_t * n

values, _ = theano.scan(
     fn = inside_loop,
     sequences=[x],
     outputs_info=[T.zeros(1)],
     non_sequences=[n],
     n_steps=x.shape[0])

sum_of_n_times_x = values[-1]
```

Loops and recurrent models

Line by line:

```
def inside_loop(x_t, acc, n):
  return acc + x_t * n
```

- This function is called at each iteration
- It takes the arguments in this order:
  1. Sequences (default: `seq[t]`)
  2. Outputs (default: `out[t-1]`)
  3. Others (no indexing)
- It returns `out[t]` for each output
- There can be many sequences, many outputs and many others:

```
f(seq_0[t], seq_1[t], .., out_0[t-1], out_1[t-1], .., other_0, other_1, ..):
```

Loops and recurrent models

```
values, _ = theano.scan(
# ...
sum_of_n_times_x = values[-1]
```

`values` is the list/tensor of all outputs through time.

```
values = [ [out_0[1], out_0[2], ...],
           [out_1[1], out_1[2], ...],
           ...]
```

If there's only one output then `values = [out[1], out[2], ...]`

Loops and recurrent models

```
fn = inside_loop,
```

The loop function we saw earlier

```
sequences=[x],
```

Sequences are indexed over their **first** dimension.

Loops and recurrent models

If you want `out[t-1]` to be an input to the loop function then you need to give `out[0]`.

```
outputs_info=[T.zeros(1)],
```

If you don't want `out[t-1]` as an input to the loop, pass `None` in outputs_info:

```
outputs_info=[None, out_1[0], out_2[0], ...],
```

You can also do more advanced "tapping", i.e. get `out[t-k]`

Loops and recurrent models

```
non_sequences=[n],
```

Variables that are used inside the loop (but not indexed).

```
n_steps=x.shape[0])
```

The number of steps that the loop should do.

Note that it is possible to do a "while" loop

Loops and recurrent models

The whole thing again

```
x = T.vector('x')
n = T.scalar('n')
def inside_loop(x_t, acc, n):
  return acc + x_t * n

values, _ = theano.scan(
     fn = inside_loop,
     sequences=[x],
     outputs_info=[T.zeros(1)],
     non_sequences=[n],
     n_steps=x.shape[0])

sum_of_n_times_x = values[-1]
```

A simple RNN

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

$$\hat{y} = \mathrm{softmax}(h_T W_y + b_y)$$

```
def loop(x_t, h_tm1, W_x, W_h, b_h):
  return T.tanh(T.dot(x_t,W_x) +
                T.dot(h_tm1, W_h) +
                b_h)

values,_ = theano.scan(loop,
    [x], [T.zeros(n_hidden)], parameters)

y_hat = T.nnet.softmax(values[-1])
```

Dimshuffle and minibatches

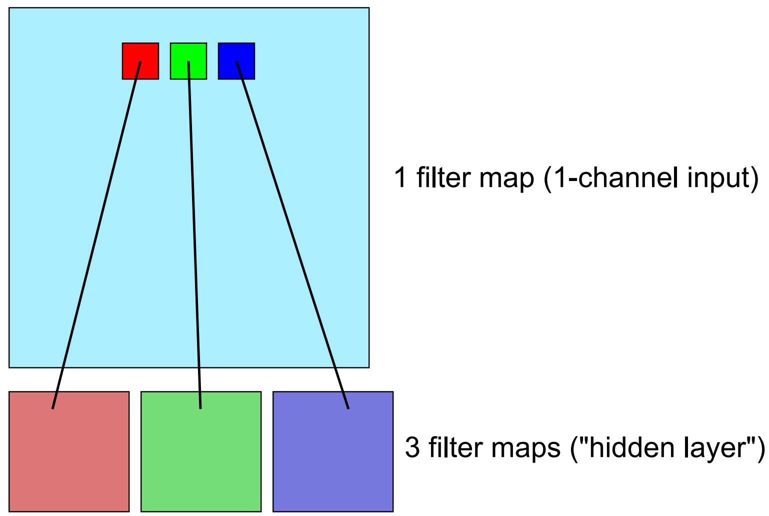
Usually you want to use minibatches ($x_{it} \in \mathbb{R}^k$):

```
# shape: (batch size, sequence length, k)
x = T.tensor3('x')
# define loop ...
v,u = theano.scan(loop,
  [x.dimshuffle(1,0,2)],
  ...)
```

This way scan iterates over the "sequence" axis.

Otherwise it would iterate over the minibatch examples.

2D convolutions



$$x : (.\,, 1, 100, 100) \quad W : (3, 1, 9, 9)$$

2D convolutions

$$\text{input } x : (m_b, n_c^{(i)}, h, w)$$
$$\text{filters } W : (n_c^{(i+1)}, n_c^{(i)}, f_s, f_s)$$

```
# x.shape: (batch size, n channels, height, width)
# W.shape: (n output channels, n input channels,
#            filter height, filter width)
output = T.nnet.conv.conv2d(x, W)
```

This convolves W with x, the output is

$$o : (m_b, n_c^{(i+1)}, h - f_s + 1, w - f_s + 1)$$

2D convolutions

Example input, 32×32 RGB images:

```
# x.shape: (batch size, n channels, height, width)
x = x.reshape((mbsize, 32, 32, 3))
x = x.dimshuffle(0,3,1,2)
# W.shape: (n output channels, n input channels,
#           filter height, filter width)
W = theano.shared(randoms((16,3,5,5)),
                  name='W-conv')
output_1 = T.nnet.conv.conv2d(x, W)
```

The flat array for an image is typically stored as a sequence of

RGBRGBRGBRGBRGBRGBRGBRGBRGB...

So you want to flip (`dimshuffle`) the dimensions so that the channels are separated.

2D convolutions

Another layer:

```
W = theano.shared(randoms((32,16,5,5)),
                  name='W-conv-2')
output_2 = T.nnet.conv.conv2d(output_1, W)
# output_2.shape: (batch size, 32, 24, 24)
```

2D convolutions

You can also do pooling:

```
from theano.tensor.downsample import max_pool_2d
# output_2.shape: (batch size, 32, 24, 24)
pooled = max_pool_2d(output_2, (2,2))
# pooled.shape: (batch size, 32, 12, 12)
```

2D convolutions

Finally, after (many) convolutions and poolings:

```
flattened = conv_output_n.flatten(ndim=2)
# then feed `flattened` to a normal hidden layer
```

we want to keep the minibatch dimension, but flatten all the other ones for our hidden layer, thus the `ndim=2`

A few tips: make classes

Make reusable classes for layers, or parts of your model:

```
class HiddenLayer:
  def __init__(self, x, n_in, n_hidden):
    self.W = shared(...)
    self.b = shared(...)
    self.output = activation(T.dot(x,W)+b)
```

A few tips: save often

It's really easy with theano/python to save and reload data:

```
class HiddenLayer:
  def __init__(self, x, n_in, n_hidden):
    # ...
    self.params = [self.W, self.b]
  def save_params(self):
     return [i.get_value() for i in self.params]
  def load_params(self, values):
     for p, value in zip(self.params, values):
       p.set_value(value)
```

A few tips: save often

It's really easy with theano/python to save and reload data:

```
import cPickle as pickle
# save
pickle.dump(model.save_params(),
            file('model_params.pkl', 'w'))
# load
model.load_params(
  pickle.load(
    file('model_params.pkl','r')))
```

You can even save whole models and functions with `pickle` but that requires a few additional tricks.

A few tips: error messages

```
ValueError: GpuElemwise. Input dimension mis-match.  Input 1 (indices
        start at 0) has shape[1] == 256, but the output's size on that axis is 128.
Apply node that caused the error: GpuElemwise{add,no_inplace}
        (<CudaNdarrayType(float32, matrix)>,
         <CudaNdarrayType(float32, matrix)>)
Inputs types: [CudaNdarrayType(float32, matrix),
               CudaNdarrayType(float32, matrix)]
```

It tells us we're trying to add $A + B$ but $A : (n, 128), B : (n, 256)$

A few tips: floatX

Theano has a default float precision:
`theano.config.floatX`

For now GPUs can only use float32:

```
TensorType(float32, matrix) cannot store a value of dtype
float64 without risking loss of precision. If you do not mind
this loss, you can: 1) explicitly cast your data to float32,
or 2) set "allow_input_downcast=True" when calling "function".
```

A few tips: read the doc

http://deeplearning.net/software/theano/library/tensor/basic.html

MNIST

http://deeplearning.net/data/mnist/mnist.pkl.gz

Opens console

A list of things I haven't talked about

(but which you can totally search for)

- Random numbers (`T.shared_randomstreams`)
- Printing/Drawing graphs (`theano.printing`)
- Jacobians, Rop, Lop and Hessian-free
- Dealing with NaN/inf
- Extending theano (implementing Ops and types)
- Saving whole models to files (`pickle`)