

Introduction to Theano

A Fast Python Library for Modelling and Training

Pascal Lamblin

Montreal Institute for Learning Algorithms (MILA)
Université de Montréal

September 25th, 2016, Stanford



Objectives

This session will have 4 parts:

- ▶ Introduction to Theano
- ▶ Hands-on example: Logistic regression
- ▶ Hands-on example: ConvNet
- ▶ Hands-on example: LSTM

All the material is online at github.com/lamblin/bayareadschool/

Theano vision

Mathematical symbolic expression compiler

- ▶ Easy to define expressions
 - ▶ Expressions mimic NumPy's syntax and semantics
 - ▶ Support for elementary operations (not only neural layers)
- ▶ Possible to manipulate those expressions
 - ▶ Substitutions
 - ▶ Gradient, R operator
 - ▶ Stability optimizations
- ▶ Fast to compute values for those expressions
 - ▶ Speed optimizations
 - ▶ Use fast back-ends (CUDA, BLAS, custom C code)
- ▶ Tools to inspect and check for correctness

Current status

- ▶ Mature: developed and used since January 2008 (8 years old)
- ▶ Good user documentation
- ▶ Active mailing list with participants worldwide
- ▶ Many contributors from different places
- ▶ Driven hundreds of research papers
- ▶ Used to teach university classes
- ▶ Core technology for Silicon Valley start-ups
- ▶ Used for research at large companies

Theano: deeplearning.net/software/theano/

Deep Learning Tutorials: deeplearning.net/tutorial/

Related projects

Many libraries are built on top of Theano (mostly machine learning)

- ▶ Blocks
- ▶ Keras
- ▶ Lasagne
- ▶ PyMC 3
- ▶ sklearn-theano
- ▶ Platoon
- ▶ Theano-MPI
- ▶ ...

Symbolic expressions

- Declaring inputs
- Defining expressions
- Deriving gradients

Function compilation

- Compiling a Theano function
- Graph optimizations
- Graph visualization

Optimized execution

- Code generation and execution
- GPU

Advanced Topics

- Looping: the scan operation
- Debugging
- Extending Theano
- New features

Overview

Theano defines a **language**, a **compiler**, and a **library**.

- ▶ Define a symbolic expression
- ▶ Compile a function that can compute values
- ▶ Execute that function on numeric values

Symbolic inputs

Symbolic, strongly-typed inputs

```
import theano
from theano import tensor as T
x = T.vector('x')
y = T.vector('y')
```

- ▶ All Theano variables have a type
- ▶ For instance ivector, fmatrix, dtensor4
- ▶ ndim, dtype, broadcastable pattern, device are part of the type
- ▶ shape and memory layout (strides) are **not**

Shared variables

```
import numpy as np
np.random.seed(42)
W_val = np.random.randn(4, 3)
b_val = np.ones(3)

W = theano.shared(W_val)
b = theano.shared(b_val)
W.name = 'W'
b.name = 'b'
```

- ▶ Symbolic variables, with a **value** associated to them
- ▶ The value is **persistent** across function calls
- ▶ The value is **shared** among all functions
- ▶ The value can be **updated**

Build an expression

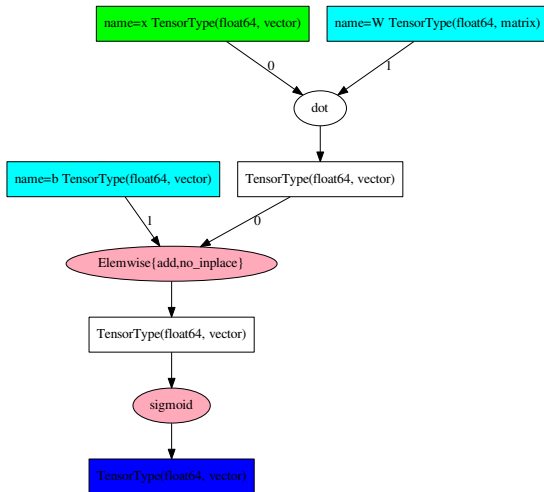
NumPy-like syntax

```
dot = T.dot(x, W)  
out = T.nnet.sigmoid(dot + b)
```

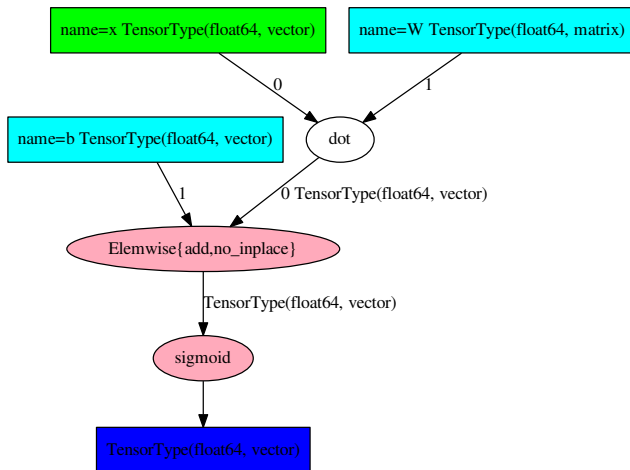
```
C = ((out - y) ** 2).sum()
```

- ▶ This creates new *variables*
- ▶ Outputs of mathematical operations
- ▶ Graph structure connecting them

`pydotprint(out, compact=False)`



pydotprint(out)



The back-propagation algorithm

Application of the chain-rule for functions from \mathbb{R}^N to \mathbb{R} .

- ▶ $C : \mathbb{R}^N \rightarrow \mathbb{R}$
- ▶ $f : \mathbb{R}^M \rightarrow \mathbb{R}$
- ▶ $g : \mathbb{R}^N \rightarrow \mathbb{R}^M$
- ▶ $C(x) = f(g(x))$
- ▶ $\frac{\partial C}{\partial x} \Big|_x = \frac{\partial f}{\partial g} \Big|_{g(x)} \cdot \frac{\partial g}{\partial x} \Big|_x$

The whole $M \times N$ Jacobian matrix $\frac{\partial g}{\partial x} \Big|_x$ is not needed.

We only need $\nabla_{g_x} : \mathbb{R}^M \rightarrow \mathbb{R}^N, v \mapsto v \cdot \frac{\partial g}{\partial x} \Big|_x$

This is implemented for (almost) each mathematical operation in Theano.

Using theano.grad

theano.grad traverses the graph, applying the chain rule.

```
dC_dW = theano.grad(C, W)
```

```
dC_db = theano.grad(C, b)
```

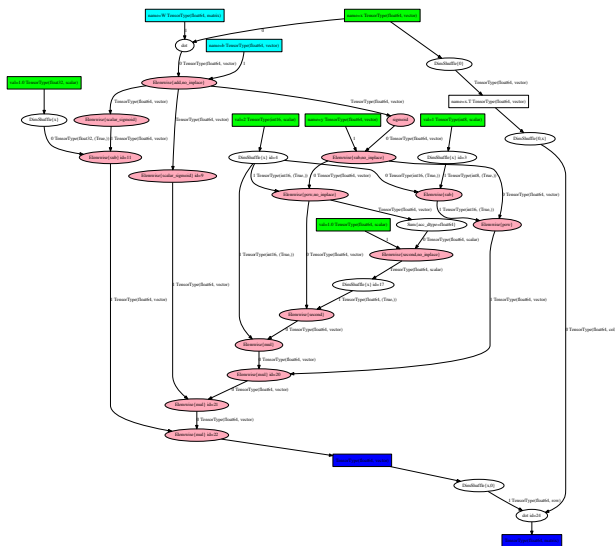
```
# or dC_dW, dC_db = theano.grad(C, [W, b])
```

- ▶ dC_{dW} and dC_{db} are symbolic expressions, like W and b
- ▶ There are no numerical values at this point
- ▶ They are part of the same computation graph
- ▶ They can also be used to build new expressions

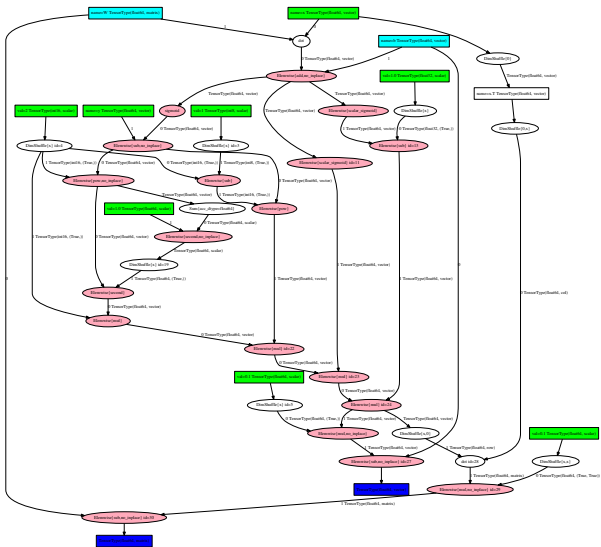
```
upd_W = W - 0.1 * dC_dW
```

```
upd_b = b - 0.1 * dC_db
```

```
pydotprint([dC_dW, dC_db])
```



```
pydotprint([upd_W, upd_b])
```



Symbolic expressions

- Declaring inputs
- Defining expressions
- Deriving gradients

Function compilation

- Compiling a Theano function
- Graph optimizations
- Graph visualization

Optimized execution

- Code generation and execution
- GPU

Advanced Topics

- Looping: the scan operation
- Debugging
- Extending Theano
- New features

Computing values

Build a callable that compute outputs given inputs

- ▶ Shared variables are implicit inputs

```
predict = theano.function([x], out)
x_val = np.random.rand(4)
print(predict(x_val))
# -> array([ 0.9421594 ,  0.73722395,  0.67606977])

monitor = theano.function([x, y], [out, C])
y_val = np.random.uniform(size=3)
print(monitor(x_val, y_val))
# -> [array([ 0.9421594 ,  0.73722395,  0.67606977]),
#      array(0.6191236997823024)]

error = theano.function([out, y], C)
print(error([0.942, 0.737, 0.676], y_val))
# -> array(0.6002210054210885)
```

Updating shared variables

A function can compute new values for shared variables, and perform updates.

```
train = theano.function([x, y], C,  
                        updates=[(W, upd_W),  
                                (b, upd_b)])  
  
print(b.get_value())  
# -> [ 1.  1.  1.]  
train(x_val, y_val)  
print(b.get_value())  
# -> [ 0.9967082  0.99340064  0.97245715]
```

- ▶ Variables W and b are **implicit inputs**
- ▶ Expressions upd_W and upd_b are **implicit outputs**
- ▶ All outputs, including the update expressions, are computed **before** the updates are performed

Graph optimizations

An optimization replaces a part of the graph with different nodes

- ▶ The types of the replaced nodes have to match
- ▶ The values should be equivalent

Different goals for optimizations:

- ▶ Merge equivalent computations
- ▶ Simplify expressions: x/x becomes 1
- ▶ Numerical stability: " $\log(1 + x)$ " becomes " $\log1p(x)$ "
- ▶ Insert in-place and destructive versions of operations
- ▶ Use specialized, efficient versions (Elemwise loop fusion, BLAS, cuDNN)
- ▶ Shape inference
- ▶ Constant folding
- ▶ Transfer to GPU

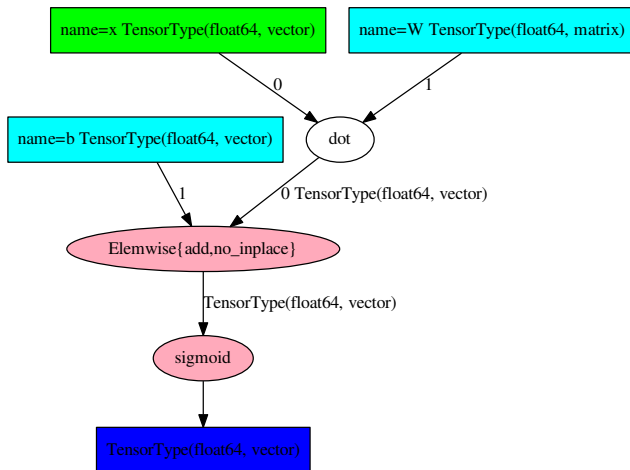
Enabling/disabling optimizations

Trade-off between compilation speed, execution speed, error detection.

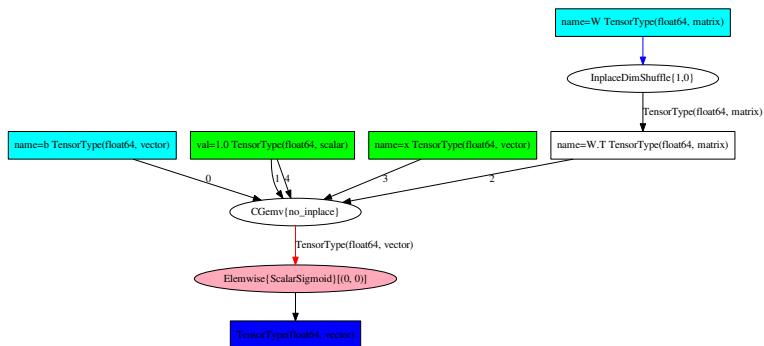
Different pre-defined modes govern the runtime and how much optimizations are applied

- ▶ `mode='FAST_RUN'`: default, make the runtime as fast as possible, launching overhead. Includes moving computation to GPU if a GPU was selected
- ▶ `mode='FAST_COMPILE'`: minimize launching overhead, around NumPy speed
- ▶ `optimizer='fast_compile'`: enables code generation and GPU use, but limits graph optimizations
- ▶ `mode='DEBUG_MODE'`: checks and double-checks everything, extremely slow
- ▶ Enable and disable particular optimizations or sets of optimizations
- ▶ Can be done globally, or for each function

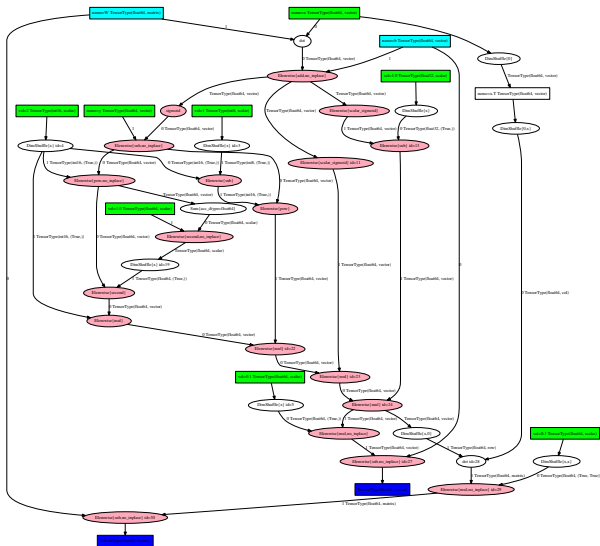
pydotprint(out)



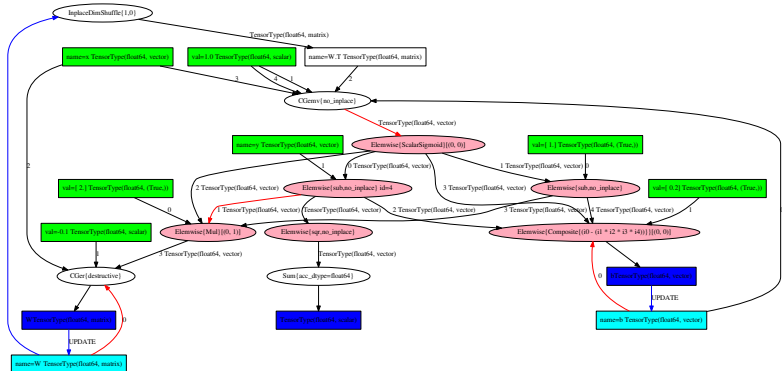
pydotprint(predict)



pydotprint([upd_W, upd_b])



pydotprint(train)



debugprint

```
debugprint(out)
```

```
sigmoid [id A] ''
|Elemwise{add,no_inplace} [id B] ''
|dot [id C] ''
| |x [id D]
| |W [id E]
|b [id F]
```

```
debugprint(predict)
```

```
Elemwise{ScalarSigmoid}[(0, 0)] [id A] '' 2
|CGemv{no_inplace} [id B] '' 1
|b [id C]
|TensorConstant{1.0} [id D]
|InplaceDimShuffle{1,0} [id E] 'W.T' 0
| |W [id F]
|x [id G]
|TensorConstant{1.0} [id D]
```

Symbolic expressions

- Declaring inputs
- Defining expressions
- Deriving gradients

Function compilation

- Compiling a Theano function
- Graph optimizations
- Graph visualization

Optimized execution

- Code generation and execution
- GPU

Advanced Topics

- Looping: the scan operation
- Debugging
- Extending Theano
- New features

Code generation and execution

Code generation for Ops:

- ▶ Ops can define C++/CUDA code computing its output values
- ▶ Dynamic code generation is possible
 - ▶ For instance, loop fusion for arbitrary sequence of element-wise operations
- ▶ Code gets compiled into a Python module, cached, and imported
- ▶ Otherwise, fall back to a Python implementation

Code execution through a runtime environment, or VM:

- ▶ Calls the functions performing computation for the Ops
- ▶ Deals with ordering constraints, lazy execution
- ▶ A C++ implementation (CVM) to avoid context switches (in/out of the Python interpreter)

Using the GPU

We want to make the use of GPUs as transparent as possible.
Theano features a new GPU back-end, with

- ▶ More dtypes, not only float32
- ▶ Experimental support for 'float16' for storage
- ▶ Easier interaction with GPU arrays from Python
- ▶ Multiple GPUs and multiple streams
- ▶ **In the development version only, and future 0.9 release**

Select GPU by setting the device flag to 'cuda' or 'cuda{0,1,2,...}'.

- ▶ All **shared** variables will be created in GPU memory
- ▶ Enables optimizations moving supported operations to GPU
- ▶ You want to make sure to use float32 for speed

Configuration flags

Configuration flags can be set in a couple of ways:

- ▶ In the `.theanorc` configuration file:

```
[global]
device = cuda0
floatX = float32
```

- ▶ `THEANO_FLAGS=device=cuda0,floatX=float32` in the shell
- ▶ In Python:

```
theano.config.floatX = 'float32'
(theano.config.device cannot be set once Theano is imported)
```

Symbolic expressions

- Declaring inputs
- Defining expressions
- Deriving gradients

Function compilation

- Compiling a Theano function
- Graph optimizations
- Graph visualization

Optimized execution

- Code generation and execution
- GPU

Advanced Topics

- Looping: the scan operation
- Debugging
- Extending Theano
- New features

Overview of scan

Symbolic looping

- ▶ Can perform map, reduce, reduce and accumulate, ...
- ▶ Can access outputs at previous time-step, or further back
- ▶ Symbolic number of steps
- ▶ Symbolic stopping condition (behaves as do ... while)
- ▶ Actually embeds a small Theano function
- ▶ Gradient through scan implements backprop through time
- ▶ Can be transferred to GPU

We will see a use of scan in the LSTM example.

Example: Loop with accumulation

```
k = T.iscalar("k")
A = T.vector("A")

# Symbolic description of the result
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)

# We only care about A**k, but scan has provided us with A**1 through A**k.
# Discard the values that we don't care about. Scan is smart enough to
# notice this and not waste memory saving them.
final_result = result[-1]

# compiled function that returns A**k
power = theano.function(inputs=[A, k], outputs=final_result, updates=updates)

print(power(range(10), 2))
# [ 0.  1.  4.  9. 16. 25. 36. 49. 64. 81.]
print(power(range(10), 4))
# [ 0.00000000e+00  1.00000000e+00  1.60000000e+01  8.10000000e+01
#  2.56000000e+02  6.25000000e+02  1.29600000e+03  2.40100000e+03
#  4.09600000e+03  6.56100000e+03]
```

Visualization, debugging, and diagnostic tools

The *definition* of a Theano function is separate from its *execution*. To help with this, we provide:

- ▶ Information in error messages
- ▶ Get information at runtime
- ▶ Monitor NaN or large value
- ▶ Test values when building the graph
- ▶ Detect common sources of slowness
- ▶ Self-diagnostic tools

Extending Theano

Theano can be extended in a few different ways

- ▶ Creating an Op with Python code
 - ▶ Easy, using Python bindings for specialized libraries (PyCUDA, ...)
 - ▶ Some runtime overhead is possible
 - ▶ Example: 3D convolution using FFT on GPU
- ▶ Creating an Op with C or CUDA code
 - ▶ Use the C-API of Python / NumPy / GpuArray, manage refcounts
 - ▶ No overhead of Python function calls, or from the interpreter
 - ▶ C code inline or in a separate file
 - ▶ Example: Caffe-style convolutions, using GEMM, on CPU and GPU
- ▶ Adding an optimization
 - ▶ Perform additional graph simplifications
 - ▶ Replace part of the graph by a new optimized Op

Features recently added to Theano

- ▶ New GPU back-end (dev branch), with:
 - ▶ Arrays of all dtypes, half-precision float (float16) for storage
 - ▶ Support for multiple GPUs in the same function
- ▶ Performance improvements
 - ▶ Better interface and implementations for convolution and transposed convolution
 - ▶ Integration of CuDNN (now v5) for 2D/3D convolutions and pooling
 - ▶ CNMeM and a similar allocator for GPU memory
 - ▶ Data-parallelism with Platoon (github.com/mila-udem/platoon/)
- ▶ Faster graph optimization phase
 - ▶ Execution of un-optimized graph on GPU (quicker compile time)
 - ▶ Various ways to avoid recompilation
- ▶ New diagnostic tools
 - ▶ Interactive visualization (d3viz)
 - ▶ PdbBreakPoint

What to expect in the near future

- ▶ Better support for int operations on GPU (indexing, argmax)
- ▶ More CuDNN operations (basic RNNs, batch normalization)
- ▶ Better support for 3D convolution / pooling
- ▶ Simpler, faster optimization mode
- ▶ Data-parallelism across nodes in Platoon

Acknowledgements

- ▶ All people working or having worked at the MILA (previously LISA), especially Theano contributors
 - ▶ Frédéric Bastien, Yoshua Bengio, James Bergstra, Arnaud Bergeron, Philemon Brakel, Olivier Breuleux, Pierre Luc Carrier, Mathieu Germain, Ian Goodfellow, Simon Lefrançois, Razvan Pascanu, Joseph Turian, David Warde-Farley, and many more
- ▶ Compute Canada, Calcul Québec, NSERC, the Canada Research Chairs, and CIFAR for providing funding or access to compute resources
- ▶ Stanford AI Lab and the organizers of the Deep Learning School

Thanks for your attention

Questions, comments, requests?

Thanks for your attention

Questions, comments, requests?

github.com/lamblin/bayareadlschool/

- ▶ Slides: [intro_theano.pdf](#)
- ▶ Companion notebook: [intro_theano.ipynb](#)

Thanks for your attention

Questions, comments, requests?

github.com/lamblin/bayareadlschool/

- ▶ Slides: [intro_theano.pdf](#)
- ▶ Companion notebook: [intro_theano.ipynb](#)

More resources

- ▶ Documentation: deeplearning.net/software/theano/
- ▶ Code: github.com/Theano/Theano/
- ▶ Article: The Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions", arxiv.org/abs/1605.02688
- ▶ Deep Learning Tutorials: deeplearning.net/tutorial/

Examples

Tutorial repository on GitHub:

`github.com/lamblin/bayareadlschool/`

- ▶ Install the dependencies
- ▶ Clone the repository
`git clone https://github.com/lamblin/bayareadlschool.git`
- ▶ Launch the notebook
`jupyter notebook bayareadlschool`
- ▶ **Logistic regression:** Open `logistic_regression.ipynb`
- ▶ **ConvNet:** Navigate to `convnet`, then `exercises.ipynb`
- ▶ **LSTM:** Navigate to `lstm`, then `exercises.ipynb`