Introduction to Theano

A Fast Python Library for Modelling and Training

Pascal Lamblin
Institut des algorithmes d'apprentissage de Montréal
Montreal Institute for Learning Algorithms
Université de Montréal

September 14th, 2016, Montréal

Objectives

This session will have 3 parts:

- Introduction to Theano (theoretical)
- Hands-on exercises on the basics of Theano
- Debugging in Theano, with exercises

All the material is online at
`github.com/lamblin/ccw_tutorial/Theano_A2016/`

Next tutorials will be:

- Sept. 21st – *Clusters*, by Mathieu Germain
- Sept. 28th – *Advanced Theano*, by Arnaud Bergeron
- Oct. 5th – *Scan: loops in Theano*, by Frédéric Bastien

Theano vision

Mathematical symbolic expression compiler

- Easy to define expressions
  - Expressions mimic NumPy's syntax and semantics
- 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: Theano has been developed and used since January 2008 (8 years old)
- Driven hundreds of research papers
- Good user documentation
- Active mailing list with participants worldwide
- Core technology for Silicon Valley start-ups
- Many contributors from different places
- Used to teach university classes
- Has been 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
- ...

Basic usage

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

Defining an expression

- Symbolic, strongly-typed inputs

```python
import theano
from theano import tensor as T
x = T.vector('x')
W = T.matrix('W')
b = T.vector('b')
```

- NumPy-like syntax to build expressions

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

Graph visualization (1)

```
debugprint(dot)
dot [id A] ''
 |x [id B]
 |W [id C]

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

Compiling a Theano function

Build a callable that compute outputs given inputs

```
f = theano.function(inputs=[x, W], outputs=dot)
g = theano.function([x, W, b], out)
h = theano.function([x, W, b], [dot, out])
i = theano.function([x, W, b], [dot + b, out])
```

Graph visualization (2)

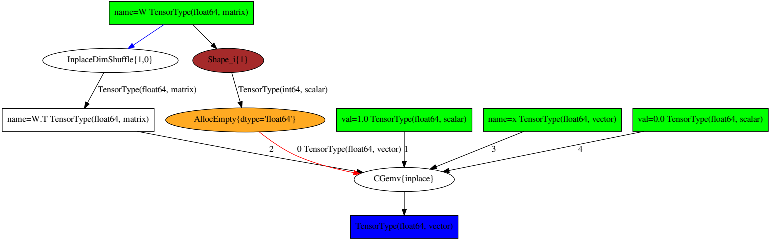
```
theano.printing.debugprint(f)
CGemv{inplace} [id A] ''   3
 |AllocEmpty{dtype='float64'} [id B] ''   2
 | |Shape_i{1} [id C] ''   1
 |   |W [id D]
 |TensorConstant{1.0} [id E]
 |InplaceDimShuffle{1,0} [id F] 'W.T'   0
 | |W [id D]
 |x [id G]
 |TensorConstant{0.0} [id H]

theano.printing.pydotprint(f)
```

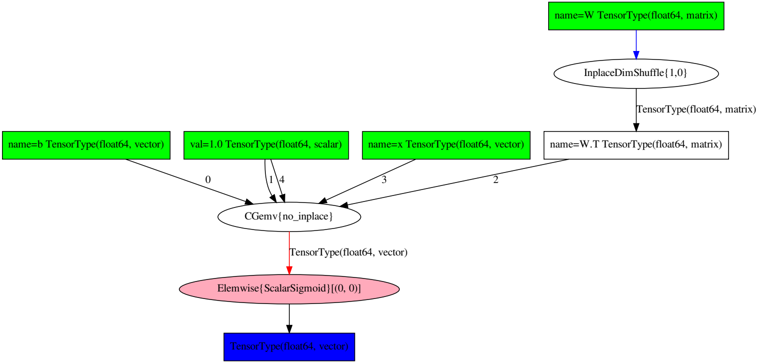
```
theano.printing.debugprint(g)
Elemwise{ScalarSigmoid}[(0, 0)] [id A] ''   2
 |CGemv{no_inplace} [id B] ''   1
   |b [id C]
   |TensorConstant{1.0} [id D]
   |InplaceDimShuffle{1,0} [id E] 'W.T'   0
   | |W [id F]
   |x [id G]
   |TensorConstant{1.0} [id D]

theano.printing.pydotprint(g)
```

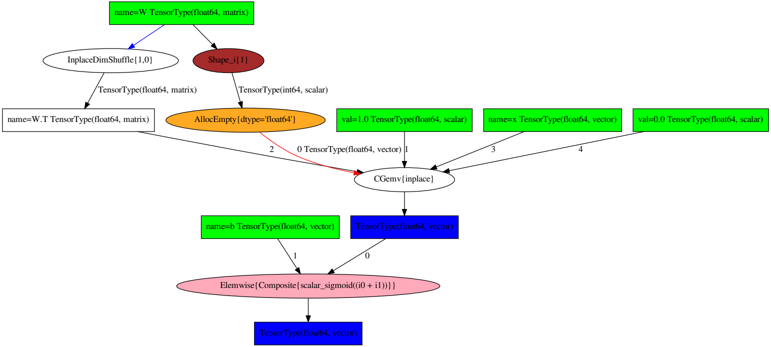
pydotprint(f)



pydotprint(g)



pydotprint(h)



d3viz

d3viz enables interactive visualization of graphs in a web browser

```python
from theano.d3viz import d3viz

d3viz(f, './d3viz_f.html')
d3viz(g, './d3viz_g.html')
d3viz(h, './d3viz_h.html')
```

Executing a Theano function

Call it with numeric values

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

f(x_val, W_val)
# -> array([ 1.79048354,  0.03158954, -0.26423186])

g(x_val, W_val, b_val)
# -> array([ 0.9421594 ,  0.73722395,  0.67606977])

h(x_val, W_val, b_val)
# -> [array([ 1.79048354,  0.03158954, -0.26423186]),
#     array([ 0.9421594 ,  0.73722395,  0.67606977])]

i(x_val, W_val, b_val)
# -> [array([ 2.79048354,  1.03158954,  0.73576814]),
#     array([ 0.9421594 ,  0.73722395,  0.67606977])]
```

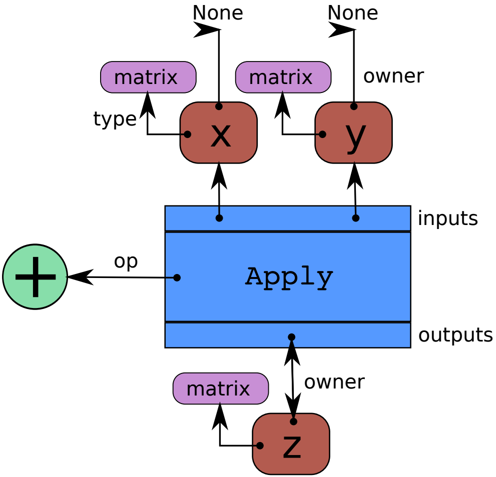
Graph structure

The graph that represents mathematical operations is **bipartite**, and has two sorts of nodes:

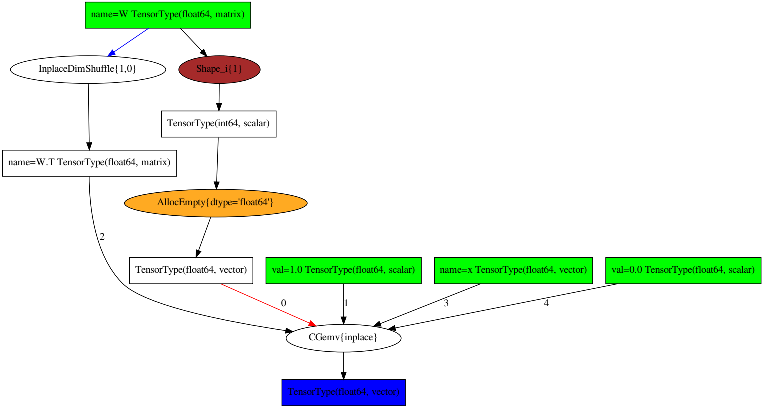
- Variable nodes, or variables, that represent *data*
- Apply nodes, that represent the application of *mathematical operations*

In practice:

- Variables are used for the graph inputs and outputs, and intermediate values
- Variables will hold data during the function execution phase
- An Apply node has inputs and outputs, which are variables
- An Apply node represents the specific application of an Op on these input variables
- The same variable can be used as inputs by several Apply nodes


None
None
matrix
matrix
owner
type
x
y
inputs
op
Apply
outputs
matrix
owner
z

pydotprint(f, compact=False)



Strong typing

- All Theano variables have a type
- Different categories of types. Most used:
  - TensorType for NumPy ndarrays
  - GpuArrayType for CUDA arrays (CudaNdarrayType in the old back-end)
  - Sparse for scipy.sparse matrices
- ndim, dtype, broadcastable pattern are part of the type
- shape and memory layout (strides) are **not**

Broadcasting tensors

- Implicit replication of arrays along broadcastable dimensions
- Broadcastable dimensions will **always** have length 1
- Such dimensions can be added to the left

```
r = T.row('r')
print(r.broadcastable)  # (True, False)
c = T.col('c')
print(c.broadcastable)  # (False, True)

f = theano.function([r, c], r + c)
print(f([[1, 2, 3]], [[.1], [.2]]))
# [[ 1.1  2.1  3.1]
#  [ 1.2  2.2  3.2]]
```

No side effects

Create new variables, cannot *change* them

- `a += 1` works, returns new variable and re-assign
- `a[:] += 1`, or `a[:] = 0` do **not** work (the `__setitem__` method cannot return a new object)
- `a = T.inc_subtensor(a[:], 1)` or `a = T.set_subtensor(a[:], 0)`
- This will create a new variable, and re-assign `a` to it
- Theano will figure out later if it can use an in-place version

Exceptions:

- The `Print()` Op
- The `Assert()` Op
- You have to re-assign (or use the returned value)
- These can disrupt some optimizations

Python keywords

We cannot redefine Python's keywords: they affect the flow when building the graph, not when executing it.

- `if var:` will always evaluate to `True`. Use `theano.ifelse.ifelse(var, expr1, expr2)`
- `for i in var:` will not work if `var` is symbolic. If `var` is numeric: loop unrolling. You can use `theano.scan`.
- `len(var)` cannot return a symbolic shape, you can use `var.shape[0]`
- `print` will print an identifier for the symbolic variable, there is a `Print()` operation

The givens keyword

With the variables defined earlier:

```
x = T.vector('x')
W = T.matrix('W')
b = T.vector('b')
dot = T.dot(x, W)
out = T.nnet.sigmoid(dot + b)
```

Substitution at the last moment, when compiling a function

```
x_ = T.vector('x_')
x_n = (x_ - x_.mean()) / x_.std()
f_n = theano.function([x_, W], dot, givens={x: x_n})
f_n(x_val, W_val)
# -> array([ 1.90651511,  0.60431744, -0.64253361])
```

Cloning with replacement

Useful when building the expression graph

```
dot_n, out_n = theano.clone(
    [dot, out],
    replace={x: (x - x.mean()) / x.std()})
f_n = theano.function([x, W], dot_n)
f_n(x_val, W_val)
# -> array([ 1.90651511,  0.60431744, -0.64253361])
```

The back-propagation algorithm

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

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

The whole $M \times N$ Jacobian matrix $\frac{\partial g}{\partial x}\Big|_x$ is not needed.
We only need $\nabla g_x : \mathbb{R}^M \to \mathbb{R}^N, v \mapsto v \cdot \frac{\partial g}{\partial x}\Big|_x$

Using theano.grad

```
y = T.vector('y')
C = ((out - y) ** 2).sum()
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

Using the gradients

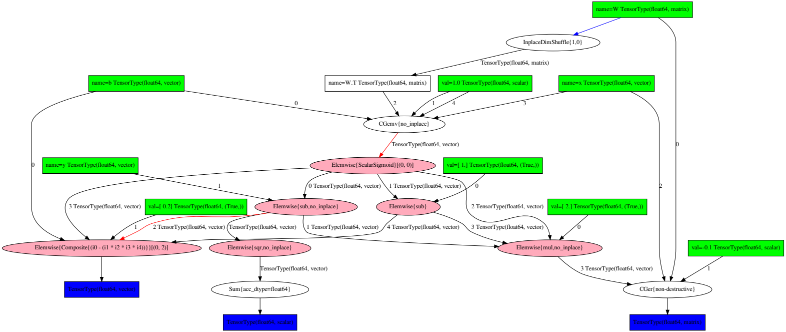
- The symbolic gradients can be used to build a Theano function

```
cost_and_grads = theano.function([x, W, b, y], [C, dC_dW, dC_db])
y_val = np.random.uniform(size=3)
print(cost_and_grads(x_val, W_val, b_val, y_val))
```

- They can also be used to build new expressions

```
upd_W = W - 0.1 * dC_dW
upd_b = b - 0.1 * dC_db
cost_and_upd = theano.function([x, W, b, y], [C, upd_W, upd_b])
print(cost_and_upd(x_val, W_val, b_val, y_val))
```

pydotprint(cost_and_upd)



Update values

Simple ways to update values

```
C_val, dC_dW_val, dC_db_val = cost_and_grads(x_val, W_val, b_val, y_val)
W_val -= 0.1 * dC_dW_val
b_val -= 0.1 * dC_db_val

C_val, W_val, b_val = cost_and_upd(x_val, W_val, b_val, y_val)
```

- Cumbersome
- Inefficient: memory, GPU transfers

Shared variables

- Symbolic variables, with a **value** associated to them
- The value is **persistent** across function calls
- The value is **shared** among all functions
- The variable has to be an **input variable**
- The variable is an **implicit input** to all functions using it

Using shared variables

```
x = T.vector('x')
y = T.vector('y')
W = theano.shared(W_val)
b = theano.shared(b_val)
dot = T.dot(x, W)
out = T.nnet.sigmoid(dot + b)
f = theano.function([x], dot)  # W is an implicit input
g = theano.function([x], out)  # W and b are implicit inputs
print(f(x_val))
# [ 1.79048354  0.03158954 -0.26423186]
print(g(x_val))
# [ 0.9421594   0.73722395  0.67606977]
```

- Use `W.get_value()` and `W.set_value()` to access the value later

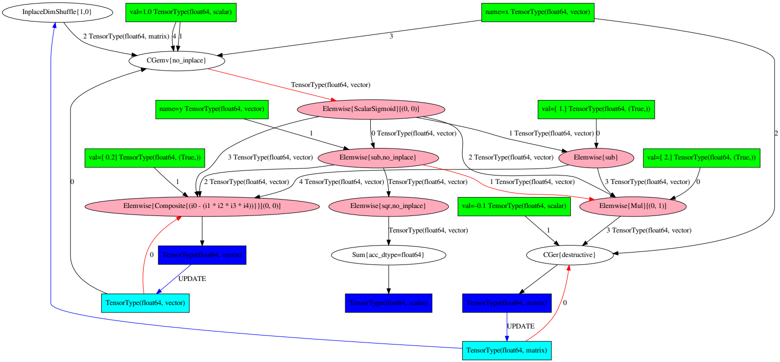
Updating shared variables

```
C = ((out - y) ** 2).sum()
dC_dW, dC_db = theano.grad(C, [W, b])
upd_W = W - 0.1 * dC_dW
upd_b = b - 0.1 * dC_db

cost_and_perform_updates = theano.function(
    inputs=[x, y],
    outputs=C,
    updates=[(W, upd_W),
             (b, upd_b)])
```

- 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

pydotprint(cost_and_perform_updates)



Graph optimizations

An optimization replaces a part of the graph with different nodes

- The types of the replaced nodes have to match

Different goals for optimizations:

- Merge equivalent computations
- Simplify expressions: x/x becomes 1
- Numerical stability: Gives the right answer for "$\log(1+x)$" even if x is really tiny.
- Insert in-place an destructive versions of operations
- Use specialized, high-performance versions (Elemwise loop fusion, GEMV, GEMM)
- 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

C code for Ops

- Each operator can define C code computing the outputs given the inputs
- Otherwise, fall back to a Python implementation

How does this work?

- In Python, build a string representing the C code for a Python module
  - Stitching together code to extract data from Python structure,
  - Takes into account input and output types (ndim, dtype, . . .)
  - String substitution for names of variables
- That module is compiled by g++
- The compiled module gets imported in Python
- Versioned cache of generated and compiled C code

For GPU code, same process, using CUDA and nvcc instead.

The C virtual machine (CVM)

A runtime environment, or VM, that calls the functions performing computation of different parts of the function (from inputs to outputs)

- Avoids context switching between C and Python
- Data structure containing
  - Addresses of inputs and outputs of all nodes (intermediate values)
  - Ordering constraints
  - Pointer to functions performing the computations
  - Information on what has been computed, and needs to be computed
- Set in advance from Python when compiling a function
- At runtime, if all operations have C code, calling the pointers will be fast
- Also enables lazy evaluation (for `ifelse` for instance)

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`
- Easier interaction with GPU arrays from Python
- Multiple GPUs and multiple streams
- **In the development version only, not the 0.8.2 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

- `'floatX'` is the default type of all tensors and sparse matrices.
- By default, aliased to `'float64'` for double precision on CPU
- Can be set to `'float32'` by a configuration flag
- You can always explicitly use `T.fmatrix()` or `T.matrix(dtype='float32')`
- Experimental support for `'float16'` on some GPUs

Configuration flags

Configuration flags can be set in a couple of ways:

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

```
theano.config.device = 'cuda0'
theano.config.floatX = 'float32'
```

- In the .theanorc configuration file:

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

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 transfered to GPU

For more about scan, see Fred's tutorial on Oct. 5th.

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

There will be a demo later during this session.

The easy way: Python

Easily wrap Python code, specialized library with Python bindings (PyCUDA, . . .)

```
import theano
import numpy
from theano.compile.ops import as_op

def infer_shape_numpy_dot(node, input_shapes):
    ashp, bshp = input_shapes
    return [ashp[:-1] + bshp[-1:]]

@as_op(itypes=[theano.tensor.fmatrix, theano.tensor.fmatrix],
       otypes=[theano.tensor.fmatrix], infer_shape=infer_shape_numpy_dot)
def numpy_dot(a, b):
   return numpy.dot(a, b)
```

- Overhead of Python call could be slow
- To define the gradient, have to actually define a class deriving from Op, and define the grad method.

Has been used to implement 3D convolution using FFT on GPU

The harder way: C code

- Understand the C-API of Python / NumPy / CudaNdarray
- Handle arbitrary strides (or use `GpuContiguous`)
- Manage refcounts for Python
- No overhead of Python function calls, or from the interpreter (if garbage collection is disabled)
- Now easier: C code in a separate file

New contributors wrote Caffe-style convolutions, using GEMM, on CPU and GPU that way.
For more about this, see Arnaud's tutorial on Sept. 28th.

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
  - Experimental support for OpenCL
- 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
  - Data-parallelism with Platoon (`github.com/mila-udem/platoon/`)
- Faster graph optimization phase
  - Execution of un-optimized graph on GPU (quicker compile time)
  - Easier serialization/deserialization of optimized function graphs, GPU shared variables
  - Swapping/removing updates without recompiling
  - Partial evaluation of a compiled function
- Diagnostic tools
  - Interactive visualization (d3viz)
  - PdbBreakPoint
  - Creation stack trace (in progress)

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, Olivier Breuleux, Pierre Luc Carrier, Ian Goodfellow, Razvan Pascanu, Joseph Turian, David Warde-Farley, Mathieu Germain, Simon Lefrançois, and many more
- Compute Canada, Calcul Québec, NSERC, the Canada Research Chairs, and CIFAR for providing funding or access to compute resources

Thanks for your attention

Questions, comments, requests?

Thanks for your attention

Questions, comments, requests?

github.com/lamblin/ccw_tutorial/Theano_A2016/

- Slides: intro_theano.pdf
- Notebook with the code examples: intro_theano.ipynb

Thanks for your attention

Questions, comments, requests?

`github.com/lamblin/ccw_tutorial/Theano_A2016/`

- Slides: `intro_theano.pdf`
- Notebook with the code examples: `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`

Exercises

Tutorial repository on GitHub:
`github.com/lamblin/ccw_tutorial/`

- Install the dependencies
- Clone the repository

  `git clone https://github.com/lamblin/ccw_tutorial.git`
- Launch the notebook

  `ipython notebook ccw_tutorial`
- Navigate to `Theano_A2016`, then `exercises.ipynb`