Hands-on Lab: Deep Learning with the Theano Python Library

Frédéric Bastien

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal
Montréal, Canada
bastienf@iro.umontreal.ca

Presentation prepared with Pierre Luc Carrier and Arnaud Bergeron

GTC 2016

Slides

- PDF of the slides: `https://goo.gl/z0tynd`
- github repo of this presentation
  `https://github.com/nouiz/gtc2016/`

High level

Python <- {NumPy/SciPy/libgpuarray} <- Theano <- {...}

- Python: OO coding language
- Numpy: n-dimensional array object and scientific computing toolbox
- SciPy: sparse matrix objects and more scientific computing functionality
- libgpuarray: GPU n-dimensional array object in C for CUDA and OpenCL
- Theano: compiler/symbolic graph manipulation

High level (2)

Many [machine learning] library build on top of Theano

- Keras
- blocks
- lasagne
- sklearn-theano: Easy deep learning by combining Theano and sklearn.
- PyMC 3
- theano-rnn
- Morb
- ...

Some models build with Theano

Some models that have been build with Theano.

- Neural Networks
- Convolutional Neural Networks
- RNN, RNN CTC, LSTM
- NADE, RNADE
- Autoencoders
- Alex Net's
- GoogleLeNet
- Overfeat
- Generative Adverserial Nets
- SVMs
- **many variations of above models and more**

Python

- General-purpose high-level OO interpreted language
- Emphasizes code readability
- Comprehensive standard library
- Dynamic type and memory management
- Easily extensible with C
- Slow execution
- Popular in *web development* and *scientific communities*

NumPy/SciPy

- NumPy provides an n-dimensional numeric array in Python
  - Perfect for high-performance computing
  - Slices of arrays are views (no copying)
- NumPy provides
  - Elementwise computations
  - Linear algebra, Fourier transforms
  - Pseudorandom number generators (many distributions)
- SciPy provides lots more, including
  - Sparse matrices
  - More linear algebra
  - Solvers and optimization algorithms
  - Matlab-compatible I/O
  - I/O and signal processing for images and audio

What's missing?

- Non-lazy evaluation (required by Python) hurts performance
- Bound to the CPU
- Lacks symbolic or automatic differentiation
- No automatic speed and stability optimization

Goal of the stack

Fast to develop
Fast to run

Description

High-level domain-specific language for numeric computation.

- Syntax as close to NumPy as possible
- Compiles most common expressions to C for CPU and/or GPU
- Limited expressivity means more opportunities for optimizations
  - Strongly typed -> compiles to C
  - Array oriented -> easy parallelism
  - Support for looping and branching in expressions
  - No subroutines -> global optimization
- Automatic speed and numerical stability optimizations

Description (2)

- Automatic differentiation and R op (Hessian Free Optimization)
- Sparse matrices (CPU only)
- Can reuse other technologies for best performance
  - BLAS, SciPy, CUDA, PyCUDA, Cython, Numba, ...
- Extensive unit-testing and self-verification
- Extensible (You can create new operations as needed)
- Works on Linux, OS X and Windows

Project status?

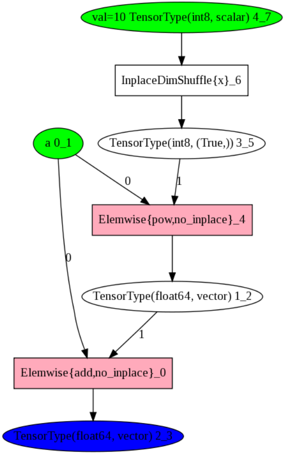
- Mature: Theano has been developed and used since January 2008 (8 yrs old)
- Driven hundreds research papers
- Good user documentation
- Active mailing list with participants from outside our institute
- Core technology for Silicon-Valley start-ups
- Many contributors (some from outside our institute)
- Used to teach many university classes
- Has been used for research at big compagnies

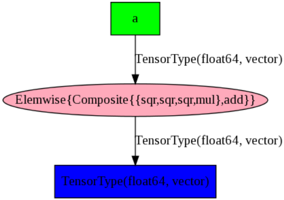
Theano: `deeplearning.net/software/theano/`
Deep Learning Tutorials: `deeplearning.net/tutorial/`

Simple example

```
import theano
# declare symbolic variable
a = theano.tensor.vector("a")

# build symbolic expression
b = a + a ** 10

# compile function
f = theano.function([a], b)

# Execute with numerical value
print f([0, 1, 2])
# prints `array([0, 2, 1026])`
```

Simple example





Overview of library

Theano is many things

- Language
- Compiler
- Python library

Scalar math

Some example of scalar operations:

```
import theano
from theano import tensor as T
x = T.scalar()
y = T.scalar()
z = x + y
w = z * x
a = T.sqrt(w)
b = T.exp(a)
c = a ** b
d = T.log(c)
```

Vector math

```
from theano import tensor as T
x = T.vector()
y = T.vector()
# Scalar math applied elementwise
a = x * y
# Vector dot product
b = T.dot(x, y)
# Broadcasting (as NumPy, very powerful)
c = a + b
```

Matrix math

```
from theano import tensor as T
x = T.matrix()
y = T.matrix()
a = T.vector()
# Matrix-matrix product
b = T.dot(x, y)
# Matrix-vector product
c = T.dot(x, a)
```

Tensors

Using Theano:

- Dimensionality defined by length of "broadcastable" argument
- Can add (or do other elemwise op) two tensors with same dimensionality
- Duplicate tensors along broadcastable axes to make size match

```
from theano import tensor as T
tensor3 = T.TensorType(
    broadcastable=(False, False, False),
    dtype='float32')
x = T.tensor3()
```

Reductions

```
from theano import tensor as T
tensor3 = T.TensorType(
    broadcastable=(False, False, False),
    dtype='float32')
x = tensor3()

total = x.sum()
marginals = x.sum(axis=(0, 2))
mx = x.max(axis=1)
```

Dimshuffle

```
from theano import tensor as T
tensor3 = T.TensorType(
    broadcastable=(False, False, False))
x = tensor3()

y = x.dimshuffle((2, 1, 0))
a = T.matrix()

b = a.T
# Same as b
c = a.dimshuffle((0, 1))

# Adding to larger tensor
d = a.dimshuffle((0, 1, 'x'))
e = a + d
```

Indexing

As NumPy! This mean slices and index selection return view

```
# return views, supported on GPU
a_tensor[int]
a_tensor[int, int]
a_tensor[start:stop:step, start:stop:step]
a_tensor[::-1] # reverse the first dimension

# Advanced indexing, return copy
a_tensor[an_index_vector] # Supported on GPU
a_tensor[an_index_vector, an_index_vector]
a_tensor[int, an_index_vector]
a_tensor[an_index_tensor, ...]
```

Compiling and running expression

- theano.function
- shared variables and updates
- compilation modes

theano.function

```
>>> from theano import tensor as T
>>> x = T.scalar()
>>> y = T.scalar()
>>> from theano import function
>>> # first arg is list of SYMBOLIC inputs
>>> # second arg is SYMBOLIC output
>>> f = function([x, y], x + y)
>>> # Call it with NUMERICAL values
>>> # Get a NUMERICAL output
>>> f(1., 2.)
array(3.0)
```

Shared variables

- It's hard to do much with purely functional programming
- "shared variables" add just a little bit of imperative programming
- A "shared variable" is a buffer that stores a numerical value for a Theano variable
- Can write to as many shared variables as you want, once each, at the end of the function
- Can modify value outside of Theano function with get_value() and set_value() methods.

Shared variable example

```
>>> from theano import shared
>>> x = shared(0.)
>>> from theano.compat.python2x import OrderedDict
>>> updates = [(x, x + 1)]
>>> f = function([], updates=updates)
>>> f()
>>> x.get_value()
1.0
>>> x.set_value(100.)
>>> f()
>>> x.get_value()
101.0
```

Compilation modes

- Can compile in different modes to get different kinds of programs
- Can specify these modes very precisely with arguments to theano.function
- Can use a few quick presets with environment variable flags

Example preset compilation modes

- FAST_RUN: default. Fastest execution, slowest compilation
- FAST_COMPILE: Fastest compilation, slowest execution. No C code.
- DEBUG_MODE: Adds lots of checks. Raises error messages in situations other modes regard as fine.
- optimizer=fast_compile: as mode=FAST_COMPILE, but with C code.
- theano.function(..., mode="FAST_COMPILE")
- THEANO_FLAGS=mode=FAST_COMPILE python script.py

Modifying expressions

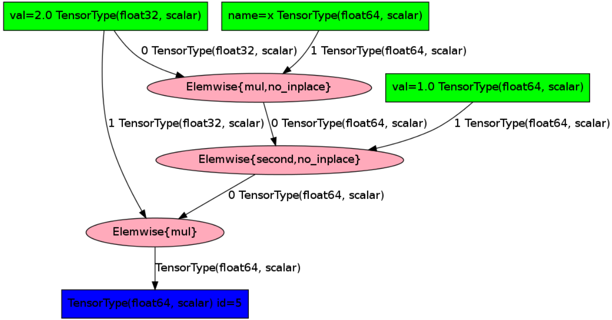
There are “macro” that automatically build bigger graph for you.

- theano.grad
- Others

Those functions can get called many times, for example to get the 2nd derivative.

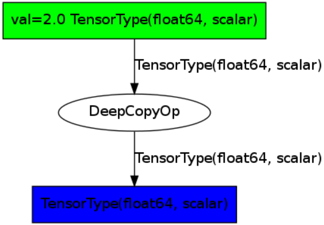
The grad method

```
>>> x = T.scalar('x')
>>> y = 2. * x
>>> g = T.grad(y, x)
# Print the not optimized graph
>>> theano.printing.pydotprint(g)
```



The grad method

```
>>> x = T.scalar('x')
>>> y = 2. * x
>>> g = T.grad(y, x)

# Print the optimized graph
>>> f = theano.function([x], g)
>>> theano.printing.pydotprint(f)
```



Others

- R_op, L_op for Hessian Free Optimization
- hessian
- jacobian
- clone the graph with replacement
- you can navigate the graph if you need (go from the result of computation to its input, recursively)

Enabling GPU

- Theano's current back-end only supports 32 bit on GPU
- libgpuarray (new-backend) supports all dtype
- CUDA supports 64 bit, but it is slow on gamer GPUs

GPU: Theano flags

Theano flags allow to configure Theano. Can be set via a configuration file or an environment variable.
To enable GPU:

- Set “device=gpu” (or a specific gpu, like “gpu0”)
- Set “floatX=float32”
- Optional: warn_float64={’ignore’, ’warn’, ’raise’, ’pdb’}

floatX

Allow to change the dtype between float32 and float64.

- T.fscalar, T.fvector, T.fmatrix are all 32 bit
- T.dscalar, T.dvector, T.dmatrix are all 64 bit
- T.scalar, T.vector, T.matrix resolve to floatX
- floatX is float64 by default, set it to float32 for GPU

CuDNN

- V3 and V4 is supported.
- It is enabled automatically if available.
- Theano flag to get an error if can't be used: "dnn.enabled=True"
- Theano flag to disable it: "dnn.enabled=False"

Debugging

- DEBUG_MODE
- Error message
- theano.printing.debugprint

Error message: code

```
import numpy as np
import theano
import theano.tensor as T
x = T.vector()
y = T.vector()
z = x + x
z = z + y
f = theano.function([x, y], z)
f(np.ones((2,)), np.ones((3,)))
```

Error message: 1st part

```
Traceback (most recent call last):
[...]
ValueError: Input dimension mis-match.
    (input[0].shape[0] = 3, input[1].shape[0] = 2)
Apply node that caused the error:
   Elemwise{add,no_inplace}(<TensorType(float64, vector)>,
                            <TensorType(float64, vector)>,
                            <TensorType(float64, vector)>)
Inputs types: [TensorType(float64, vector),
               TensorType(float64, vector),
               TensorType(float64, vector)]
Inputs shapes: [(3,), (2,), (2,)]
Inputs strides: [(8,), (8,), (8,)]
Inputs values: [array([ 1.,  1.,  1.]),
                array([ 1.,  1.]),
                array([ 1.,  1.])]
Outputs clients: [['output']]
```

Error message: 2st part

HINT: Re-running with most Theano optimization disabled could give you a back-traces when this node was created. This can be done with by setting the Theano flags "optimizer=fast_compile". If that does not work, Theano optimizations can be disabled with "optimizer=None".
HINT: Use the Theano flag "exception_verbosity=high" for a debugprint of this apply node.

Error message: traceback

```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    f(np.ones((2,)), np.ones((3,)))
  File "/u/bastienf/repos/theano/compile/function_module.py",
        line 589, in __call__
    self.fn.thunks[self.fn.position_of_error])
  File "/u/bastienf/repos/theano/compile/function_module.py",
        line 579, in __call__
    outputs = self.fn()
```

Error message: optimizer=fast_compile

```
Backtrace when the node is created:
  File "test.py", line 7, in <module>
    z = z + y
```

debugprint

```
>>> from theano.printing import debugprint
>>> debugprint(a)
Elemwise{mul,no_inplace} [id A] ''
 |TensorConstant{2.0} [id B]
 |Elemwise{add,no_inplace} [id C] 'z'
   |<TensorType(float64, scalar)> [id D]
   |<TensorType(float64, scalar)> [id E]
```

Inputs

```
# Load from disk and put in shared variable.
datasets = load_data(dataset)
train_set_x, train_set_y = datasets[0]
valid_set_x, valid_set_y = datasets[1]

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

# generate symbolic variables for input minibatch
x = T.matrix('x')  # data, 1 row per image
y = T.ivector('y')  # labels
```

Model

```python
n_in = 28 * 28
n_out = 10

# weights
W = theano.shared(
        numpy.zeros((n_in, n_out),
                    dtype=theano.config.floatX))

# bias
b = theano.shared(
        numpy.zeros((n_out,),
                    dtype=theano.config.floatX))
```

Computation

```
# the forward pass
p_y_given_x = T.nnet.softmax(T.dot(input, W) + b)

# cost we minimize: the negative log likelihood
l = T.log(p_y_given_x)
cost = -T.mean(l[T.arange(y.shape[0]), y])

# the error
y_pred = T.argmax(p_y_given_x, axis=1)
err = T.mean(T.neq(y_pred, y))
```

Gradient and updates

```
# compute the gradient of cost
g_W, g_b = T.grad(cost=cost, wrt=(W, b))

# model parameters updates rules
updates = [(W, W - learning_rate * g_W),
           (b, b - learning_rate * g_b)]
```

Training function

```
# compile a Theano function that train the model
train_model = theano.function(
    inputs=[index], outputs=(cost, err),
    updates=updates,
    givens={
        x: train_set_x[index * batch_size:
                       (index + 1) * batch_size],
        y: train_set_y[index * batch_size:
                       (index + 1) * batch_size]
    }
)
```

Inputs

```
# Load from disk and put in shared variable.
datasets = load_data(dataset)
train_set_x, train_set_y = datasets[0]
valid_set_x, valid_set_y = datasets[1]

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

x = T.matrix('x')   # the data, 1 row per image
y = T.ivector('y')  # labels

# Reshape matrix of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible for convolution
layer0_input = x.reshape((batch_size, 1, 28, 28))
```

Model

```
image_shape=(batch_size, 1, 28, 28)
filter_shape=(nkerns[0], 1, 5, 5)

W_bound = ...
W = theano.shared(
    numpy.asarray(
        rng.uniform(low=-W_bound, high=W_bound,
                    size=filter_shape),
        dtype=theano.config.floatX))

# the bias is a 1D tensor
# one bias per output feature map
b_values = numpy.zeros((filter_shape[0],),dtype=...
b = theano.shared(b_values)
```

Computation

```
# convolve input feature maps with filters
conv_out = conv.conv2d(input=x, filters=W)

# downsample each feature map individually,
# using maxpooling
pooled_out = downsample.max_pool_2d(
    input=conv_out,
    ds=(2, 2), // poolsize
    ignore_border=True)

output = T.tanh(pooled_out +
                b.dimshuffle('x', 0, 'x', 'x'))
```

ipython notebook

- Introduction
- Exercises (Theano only exercises)
- lenet (small CNN model to quickly try it)

Connection instructions

- Navigate to `nvlabs.qwiklab.com`
- Login or create a new account
- Select the "Instructor-Led Hands-on Labs" class
- Find the lab called "Theano" and click Start
- After a short wait, lab instance connection information will be shown
- Please ask Lab Assistants for help!

Further hands-on training

Check out the Self-Paced labs at the conference:

- Deep Learning, CUDA, OpenACC, Tools and more!
- Just grab a seat and take any available lab
- Located in the lower-level outside of LL20C

You will also receive Credits to take additional labs at nvidia.qwiklab.com

- Log in using the same account you used in this lab

Where to learn more

- GTC2016 sessions:
  - S6845 - Theano at a Glance: A Framework for Machine Learning
- Deep Learning Tutorials with Theano: `deeplearning.net/tutorial`
- Theano tutorial: `deeplearning.net/software/tutorial`
- Theano website: `deeplearning.net/software`
- You can also see frameworks on top of Theano like Blocks, Keras, Lasagne, …

Questions, acknowledgments

Questions?
Acknowledgments

- All people working or having worked at the LISA lab/MILA institute
- All Theano users/contributors
- Compute Canada, RQCHP, NSERC, NVIDIA, and Canada Research Chairs for providing funds, access to compute resources, hardware or GPU libraries.