Theano

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

TODO: Presentation prepared with Pierre Luc Carrier and Arnaud Bergeron TODO: slides for nvlabs, how to. TODO: lab.zip

MILA
Institut de Montréal des algorithmes d'apprentissage

GTC 2015

Université de Montréal

Slides

PDF of the slides: `http://goo.gl/tg2cKJ`

High level

Python <- {NumPy/SciPy/libgpuarray} <- Theano <- Pylearn2

- 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
- Pylearn2: machine learning framework for researchers

Python

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

NumPy/SciPy

- Python floats are full-fledged objects on the heap
  - Not suitable for high-performance computing!
- 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
  - No subroutines -> global optimization
  - Strongly typed -> compiles to C
  - Array oriented -> easy parallelism
  - Support for looping and branching in expressions
- Automatic speed and stability optimizations
- Can reuse other technologies for best performance.
  - BLAS, SciPy, Cython, Numba, PyCUDA, CUDA, ...
- Automatic differentiation and R op
- Sparse matrices (CPU only)
- Extensive unit-testing and self-verification
- Works on Linux, OS X and Windows

Project status?

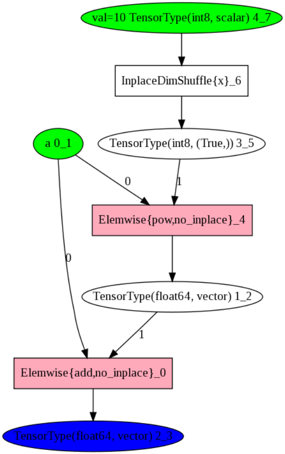
- Mature: Theano has been developed and used since January 2008 (7 yrs old)
- Driven hundreds research papers
- Good user documentation
- Active mailing list with participants from outside our lab
- Core technology for a few Silicon-Valley start-ups
- Many contributors (some from outside our lab)
- Used to teach many university classes
- Has been used for research at big compagnies

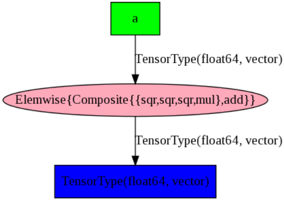
Theano: `deeplearning.net/software/theano/`
Deep Learning Tutorials: `deeplearning.net/tutorial/`

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 Language

- Operations on scalar, vector, matrix, tensor, and sparse variables
- Linear algebra
- Element-wise nonlinearities
- Convolution
- Indexing, slicing and advanced indexing.
- Reduction
- Dimshuffle (n-dim transpose)
- Extensible

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

```python
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) on 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

```python
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 all slices, 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
- TODO: optimizations

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
- Modify outside 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 = 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
```

Which dict?

- Use theano.compat.python2x.OrderedDict
- Not collections.OrderedDict
  - This isn't available in older versions of python
- Not {} aka dict
  - The iteration order of this built-in class is not deterministic (thanks, Python!) so if Theano accepted this, the same script could compile different C programs each time you run it

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

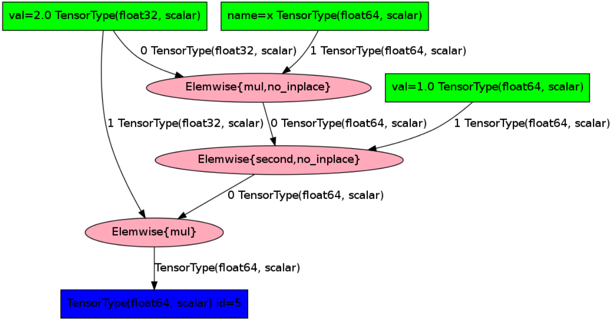
- FAST_RUN: default. Fastest execution, slowest compilation
- FAST_COMPILE: Fastest compilation, slowest execution. No C code.
- DEBUG_MODE: Adds lots of checks. Raises error messages in situations other modes regard as fine.
- optimizer=fast_compile: as mode=FAST_COMPILE, but with C code.
- theano.function(...,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

Modifying expressions

- The grad method
- Others

The grad method

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



Others

- R_op, L_op for hessian free
- hessian
- jacobian
- you can navigate the graph if you need (go from the result of computation to its input, recursively)

Enabling GPU

- Theano current back-end only supports 32 bit on GPU
- libgpuarray (new-backend) support all dtype
- CUDA supports 64 bit, but is slow on gamer GPUs
- T.fscalar, T.fvector, T.fmatrix are all 32 bit
- T.scalar, T.vector, T.matrix resolve to 32 bit or 64 bit depending on theano's floatX flag
- floatX is float64 by default, set it to float32
- Set device flag to gpu (or a specific gpu, like gpu0)
- Flag: warn_float64='ignore', 'warn', 'raise', 'pdb'

CuDNN

- R1 and R2 is supported.
- It is enabled automatically if available.
- Theano flag to get error if can't be used: "optimizer_including=cudnn"

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 scalar values: ['not scalar', 'not scalar', 'not scalar']
```

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
HINT: Use the Theano flag 'exception_verbosity=high'
for a debugprint of this apply node.
```

Error message: exception_verbosity=high

```
Debugprint of the apply node:
Elemwise{add,no_inplace} [@A] <TensorType(float64, vector)> ''
 |<TensorType(float64, vector)> [@B] <TensorType(float64, vector)>
 |<TensorType(float64, vector)> [@C] <TensorType(float64, vector)>
 |<TensorType(float64, vector)> [@C] <TensorType(float64, vector)>
```

Error message: optimizer=fast_compile

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

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()
```

debugprint

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

Scan

- Allows looping (for, map, while)
- Allows recursion (reduce)
- Allows recursion with dependency on many of the previous time steps
- Optimize some cases like moving computation outside of scan
- The Scan grad is done via Backpropagation Through Time(BPTT)

When not to use scan

- If you only need it for "vectorization" or "broadcasting". tensor and numpy.ndarray support them natively. This will be much better for that use case.
- If you do a fixed number of iteration that is very small (2,3). You are probably better to just unroll the graph to do it.

Scan Example1: Computing tanh(v.dot(W) + b) * d where b is binomial I

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variables
W = T.matrix("W")
X = T.matrix("X")
b_sym = T.vector("b_sym")

# define shared random stream
trng = T.shared_randomstreams.RandomStreams(1234)
d=trng.binomial(size=W[1].shape)
```

Scan Example1: Computing tanh(v.dot(W) + b) * d where d is binomial (2)

```
results, updates = theano.scan(
    lambda v: T.tanh(T.dot(v, W) + b_sym) * d,
    sequences=X)
f = theano.function(inputs=[X, W, b_sym],
                    outputs=[results],
                    updates=updates)
x = np.eye(10, 2, dtype=theano.config.floatX)
w = np.ones((2, 2), dtype=theano.config.floatX)
b = np.ones((2), dtype=theano.config.floatX)

print f(x, w, b)
```

Scan Example2: Computing pow(A, k)

```
import theano
import theano.tensor as T
theano.config.warn.subtensor_merge_bug = False

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

def inner_fct(prior_result, B):
    return prior_result * B
```

Scan Example2: Computing pow(A, k) (2)

```
result, updates = theano.scan(
    fn=inner_fct,
    outputs_info=T.ones_like(A),
    non_sequences=A, n_steps=k)

# Scan provide us with A ** 1 through A ** k.
# Keep only the last value. Scan optimize memory.
final = result[-1]

power = theano.function(inputs=[A, k], outputs=final,
                        updates=updates)
print power(range(10), 2)
#[  0.   1.   4.   9.  16.  25.  36.  49.  64.
81.]
```

Scan signature

```
result, updates = theano.scan(
    fn=inner_fct,
    sequences=[]
    outputs_info=[T.ones_like(A)],
    non_sequences=A,
    n_steps=k)
```

- Updates are needed if there are random numbers generated in the inner function
  - Pass them to the call theano.function(..., updates=updates)
- The inner function of scan takes arguments like this: scan: sequences, outputs_info, non sequences

Exercices

- Theano exercice: Work through the “0[1-4]*” exercices (directory):
  Available at “git clone https://github.com/nouiz/gtc2015.git”.
- Scan exercices: http://deeplearning.net/software/theano/tutorial/loop.html#exercise

Deep Learning Tutorial on LSTM:
http://deeplearning.net/tutorial/lstm.html
(It have the papers

Acknowledgments

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

Questions?