Theano and LSTM for Sentiment Analysis

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

Next.ML 2015

Laboratoire d'Informatique
des Systèmes Adaptatifs
http://www.iro.umontreal.ca/~lisa

Université de Montréal

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

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

- Mathematical symbolic expression compiler
- Expressions mimic NumPy's syntax and semantics
- Dynamic C/CUDA code generation
  - C/C++, CUDA, OpenCL, PyCUDA, Cython, Numba, ...
- Efficient symbolic differentiation
- Speed and stability optimizations
  - Gives the right answer for "$\log(1+x)$" even if x is really tiny.
- Extensive unit-testing and self-verification
- Works on Linux, OS X and Windows
- Transparent use of a GPU
  - `float32` only for now (libgpuarray provides much more)
  - Limited support on Windows
- Sparse operations (CPU only)

Overview of Library

Theano is many things

- Language
- Compiler
- Python library

Project status?

- Mature: Theano has been developed and used since January 2008 (7 yrs old)
- Driven hundreads 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/`

Overview

Theano language:

- Operations on scalar, vector, matrix, tensor, and sparse variables
- Linear algebra
- Element-wise nonlinearities
- Convolution
- Extensible

Theano

High-level domain-specific language tailored to 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 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

Overview

Using Theano:

- define expression $f(x, y) = x + y$
- compile expression

```
int f(int x, int y){
  return x + y;
}
```

- execute expression

```
>>> f(1, 2)
3
```

Building expressions

- Scalars
- Vectors
- Matrices
- Tensors
- Reduction
- Dimshuffle

Scalar math

Using Theano:

- define expression $f(x, y) = x + y$
- compile expression

```
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

```python
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
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:

- define expression $f(x, y) = x + y$
- compile expression
  - 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 = tensor3()
```

Reductions

Using Theano:

- define expression $f(x, y) = x + y$
- compile expression

```python
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,
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
```

Compiling and running expression

- theano.function
- shared variables and updates
- compilation modes
- compilation for GPU
- 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, and will limit the portability of your code
- 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. Spends a lot of time on compilation to get an executable that runs fast.
- FAST_COMPILE: Doesn't spend much time compiling. Executable usually uses python instead of compiled C code. Runs slow.
- DEBUG_MODE: Adds lots of checks. Raises error messages in situations other modes regard as fine.

Compilation for GPU

- Theano current back-end only supports 32 bit on GPU
- CUDA supports 64 bit, but is slow in gamer card
- 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)

Modifying expressions

- The grad method
- Variable nodes
- Types
- Ops
- Apply nodes

The grad method

```
>>> x = T.scalar('x')
>>> y = 2. * x
>>> g = T.grad(y, x)
>>> from theano.printing import min_informative_str
>>> print min_informative_str(g)
A. Elemwise{mul}
 B. Elemwise{second,no_inplace}
  C. Elemwise{mul,no_inplace}
   D. TensorConstant{2.0}
   E. x
  F. TensorConstant{1.0}
 <D>
```

Theano Variables

- A Variable is a theano expression
- Can come from T.scalar, T.matrix, etc.
- Can come from doing operations on other Variables
- Every Variable has a type field, identifying its Type
  e.g. TensorType((True, False), 'float32')
- Variables can be thought of as nodes in a graph

Ops

- An Op is any class that describes a mathematical function of some variables
- Can call the op on some variables to get a new variable or variables
- An Op class can supply other forms of information about the function, such as its derivatives

Apply nodes

- The Apply class is a specific instance of an application of an Op
- Notable fields:
  - op: The Op to be applied
  - inputs: The Variables to be used as input
  - outputs: The Variables produced
- Variable.owner identifies the Apply that created the variable
- Variable and Apply instances are nodes and owner/ inputs/outputs identify edges in a Theano graph

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, ve
                            <TensorType(float64, ve
                            <TensorType(float64, ve
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',
```

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, v
 |<TensorType(float64, vector)> [@B] <TensorType(flo
 |<TensorType(float64, vector)> [@C] <TensorType(flo
 |<TensorType(float64, vector)> [@C] <TensorType(flo
```

Error message: optimizer=fast_compile

```
Backtrace when the node is created:
  File "test.py", line 7, in <module>
    z = z + y
  File "/home/nouiz/src/Theano/theano/tensor/var.py'
    return theano.tensor.basic.add(self, other)
```

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_m
       line 589, in __call__
    self.fn.thunks[self.fn.position_of_error])
  File "/u/bastienf/repos/theano/compile/function_m
       line 579, in __call__
    outputs = self.fn()
```

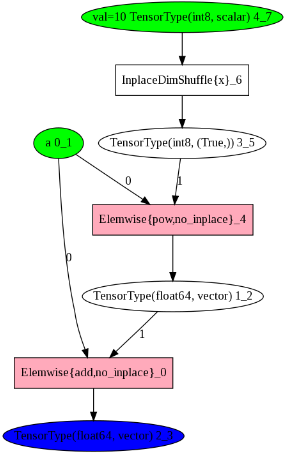
debugprint

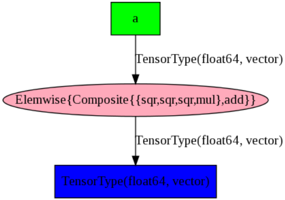
```
>>> from theano.printing import debugprint
>>> debugprint(a)
Elemwise{mul,no_inplace} [@A] ''
 |TensorConstant{2.0} [@B]
 |Elemwise{add,no_inplace} [@C] 'z'
   |<TensorType(float64, scalar)> [@D]
   |<TensorType(float64, scalar)> [@E]
```

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)
print f([0, 1, 2])
# prints `array([0, 2, 1026])`
```

Simple example: graph optimization





libgpuarray: Design Goals

- Have the base object in C to allow collaboration with more projects.
  - We want people from C, C++, ruby, R, ... all use the same base GPU ndarray.
- Be compatible with CUDA and OpenCL.
- Not too simple, (don't support just matrix).
- Support all dtype.
- Allow strided views.
- But still easy to develop new code that support only a few memory layout.
  - This ease the development of new code.

Conclusion

Theano/Pylearn2/libgpuarry provide an environment for machine learning that is: **Fast to develop**
Fast to run

Exercices

Work through the “01_buildbing_expressions” directory now.
Available at
“git clone https://github.com/nouiz/ccw_tutorial_theano.git”.

Acknowledgments

- All people working or having worked at the LISA lab.
- All Theano/Pylearn 2 users/contributors
- Compute Canada, RQCHP, NSERC, and Canada Research Chairs for providing funds or access to compute resources.

Questions?