Theano

A Fast Python Library for Modelling and Training

Kaushal Kishore
8th Semester, B. Tech (CSE)
Indian Institute of Technology, Palakkad
Intern - Digital Horizons

April 28th, 2020, New Delhi

theano

theano

Objectives

This tutorial will have 5 parts:

- Introduction to Theano – Motivation and design
- Symbolic Expressions
- Fuction compilation
- Optimized Execution
- Case study – Logistic Regression

Goals

Expressing models as mathematical expressions

- Not only a collection of standard layers or modules
- Not only regular gradient descent
- From an interpreted / scripting language

Automatically deriving gradients

- Define gradients for basic, elementary operations
- Treat those gradients as mathematical expressions as well
- Simplify automatically the resulting expression

Training the model efficiently

- Without having to write C / C++ / CUDA code
- Automatic simplification of the graph
- Automatic code generation

Theano: A mathematical symbolic expression compiler

Easy to define expressions

- Using Python
- 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)
- Inplace optimizations to reduce memory usage

Tools to inspect and check for correctness

Theano

Theano is a Python library that lets you to define, optimize, and evaluate mathematical expressions, especially ones with multi-dimensional arrays (`numpy.ndarray`). Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs.

Current status

- Mature: developed and used since January 2008 (12 years old)
- Theano 1.0 released in November 2017
- Driven > 1000 research papers
- Many contributors (332 for version 1.0)
- Active mailing list with participants worldwide
- 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
- rllab
- PyMC 3
- ...

For parallelism

- Platoon
- Theano-MPI
- Synkhronos
- Elephas (through Keras)

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

Sneak Peek

```
import theano
from theano import tensor
# declare two symbolic floating-point scalars
a = tensor.dscalar()
b = tensor.dscalar()
# create a simple expression
c = a + b
# convert the expression into a callable object that takes (a,b)
# values as input and computes a value for c
f = theano.function([a,b], c)
# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
assert 4.0 == f(1.5, 2.5)
```

Sneak Peek

Theano is not a programming language in the normal sense because you write a program in Python that builds expressions for Theano. Still it is like a programming language in the sense that you have to

- declare variables (a,b) and give their types
- build expressions for how to put those variables together
- compile expression graphs to functions in order to use them for computation.

It is good to think of `theano.function` as the interface to a compiler which builds a callable object from a purely symbolic graph. One of Theano's most important features is that `theano.function` can optimize a graph and even compile some or all of it into native machine instructions.

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**

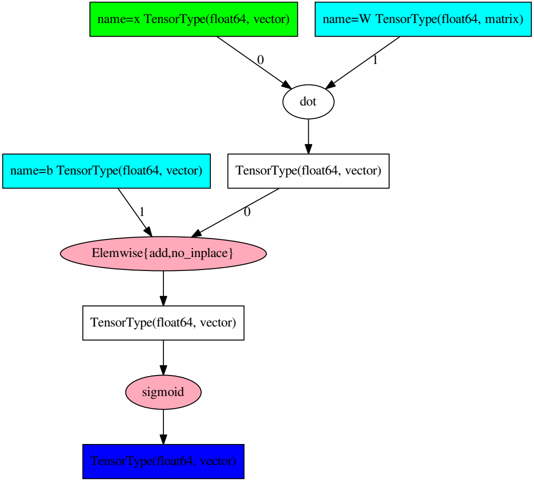
Build an expression

NumPy-like syntax

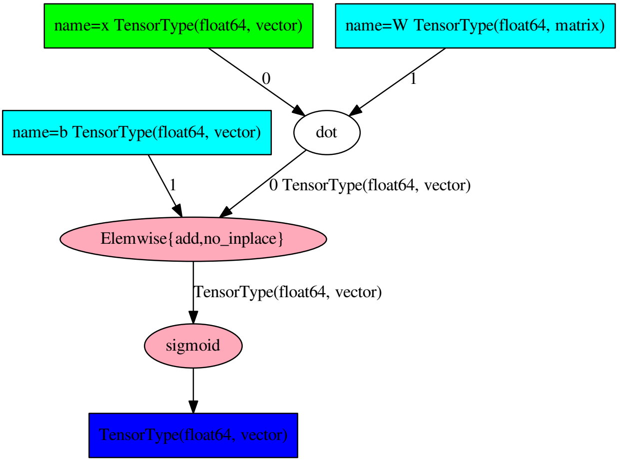
```
dot = T.dot(x, W)
out = T.nnet.sigmoid(dot + b)

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

- 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 $\mathbb{R}$.

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

Using theano.grad

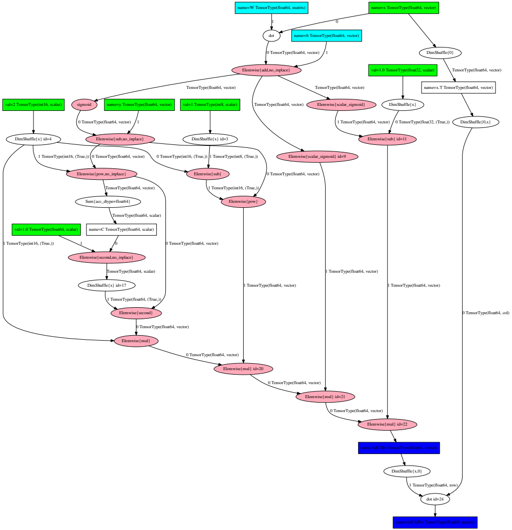
theano.grad traverses the graph, applying the chain rule.

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

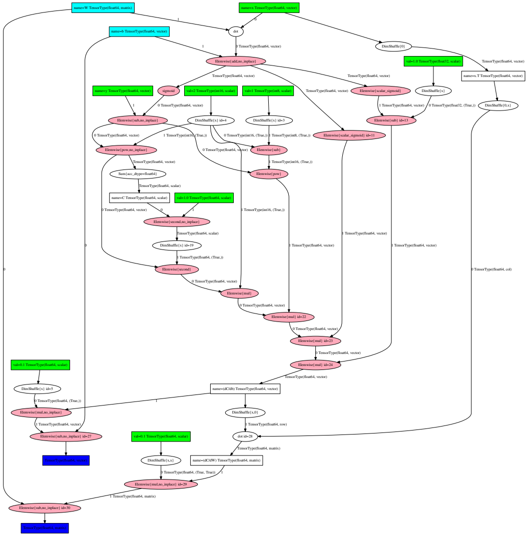
- dC_dW and dC_db are symbolic expressions, like out and C
- There are no numerical values at this point
- They are part of the same computation graph
- They can also be used to build new expressions

```
upd_W = W - 0.1 * dC_dW
upd_b = b - 0.1 * dC_db
```

pydotprint([dC_dW, dC_db])



pydotprint([upd_W, upd_b])



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(0.6137821438190066)]

error = theano.function([out, y], C)
print(error([0.942, 0.737, 0.676], y_val))
# -> array(0.613355628529845)
```

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())
# -> [ 0.99639999  0.97684097  0.98318412]
```

- 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
- Arithmetic expressions: x/x becomes 1
- Numerical stability: "$\log(1+x)$" becomes "log1p(x)"
- Insert in-place an destructive versions of operations
- Use specialized, efficient versions (Elemwise loop fusion, BLAS, cuDNN)
- Shape inference
- Constant folding
- Use of GPU for computations

http://deeplearning.net/software/theano/optimizations.html

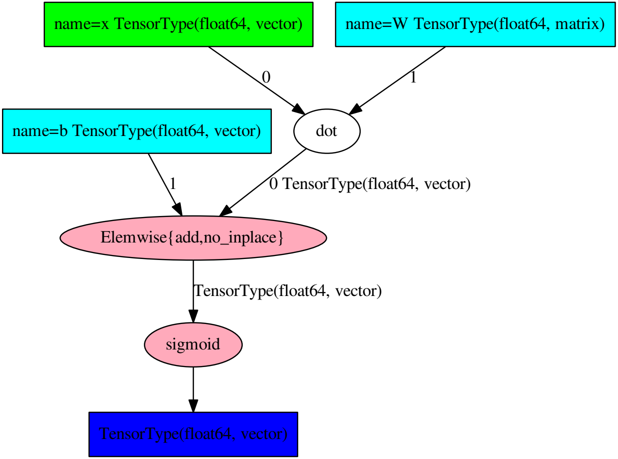
Enabling/disabling optimizations

Every time `theano.function` is called, the symbolic relationships between the input and output Theano variables are optimized and compiled. The way this compilation occurs is controlled by the value of the mode parameter.

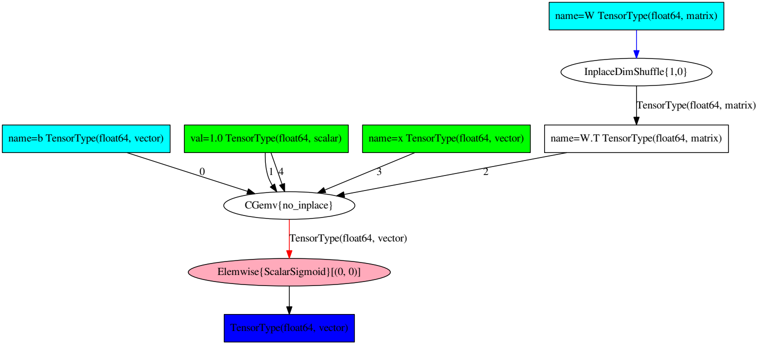
- `mode='FAST_RUN'`: Apply all optimizations and use C implementations where possible.
- `mode='FAST_COMPILE'`: Apply just a few graph optimizations and only use Python implementations. So GPU is disabled.
- `mode='DebugMode'`: Verify the correctness of all optimizations, and compare C and Python implementations. This mode can take much longer than the other modes, but can identify several kinds of problems.
- `mode='NanGuardMode'`: Same optimization as `FAST_RUN`, but check if a node generate nans.

The default mode is typically `FAST_RUN`, but it can be controlled via the configuration variable `config.mode`, which can be overridden by passing the keyword argument to `theano.function`.

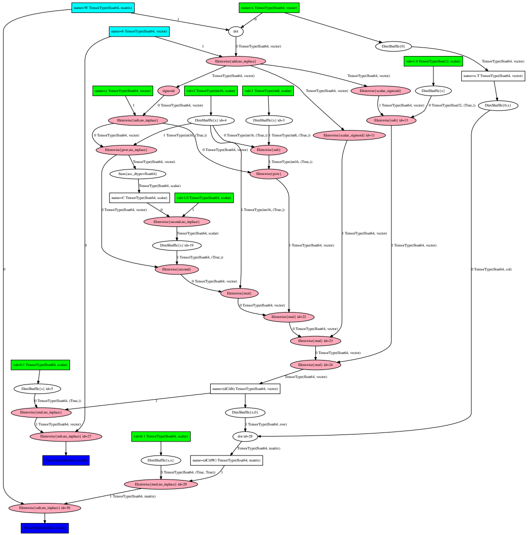
pydotprint(out)



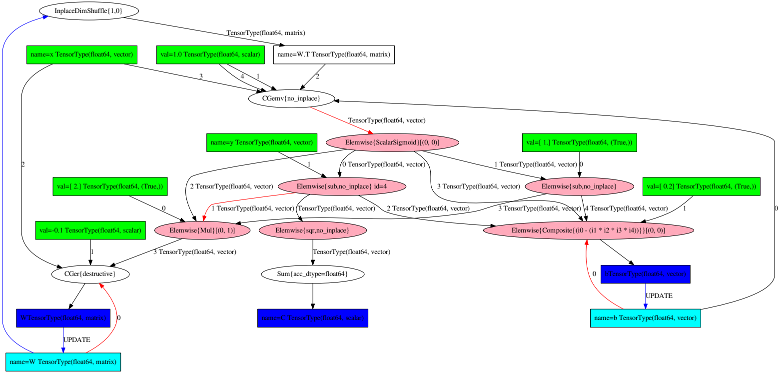
pydotprint(predict)



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]
```

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

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

- All **shared** variables will be created in GPU memory by default
- Enables optimizations moving supported operations to GPU
- Use `float32` for better GPU performance.

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, but you can call `theano.gpuarray.use('cuda0')`)

Logistic Regression-I

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400         # training sample size
feats = 784     # number of input variables

# generate a dataset: D = (input_values, target_class)
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.dmatrix("x")
y = T.dvector("y")
```

Logistic Regression-II

```
# initialize the weight vector w randomly
# weight w and bias variable b are shared
# so they keep their values between
# training iterations (updates)
w = theano.shared(rng.randn(feats), name="w")

# initialize the bias term
b = theano.shared(0., name="b")

print("Initial model:")
print(w.get_value())
print(b.get_value())
```

Logistic Regression-III

```
# Construct Theano expression graph

# Probability that target = 1
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))

# The prediction thresholded
prediction = p_1 > 0.5

# Cross-entropy loss function
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1)

# The cost to minimize
cost = xent.mean() + 0.01 * (w ** 2).sum()

# Compute the gradient of the cost w.r.t weight vector w and bias term b
gw, gb = T.grad(cost, [w, b])
```

Logistic Regression-IV

```
# Compile
train = theano.function(
          inputs=[x,y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])

print("Final model:")
print(w.get_value())
print(b.get_value())
print("target values for D:")
print(D[1])
print("prediction on D:")
print(predict(D[0]))
```

Thanks for your attention

The End!!!