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

Deep Learning Summer School
June 27th, 2017, Montréal

Université de Montréal

Objectives

This tutorial will have 4 parts:

- Introduction to Theano – Motivation and design
- Walk-through example – LeNet on MNIST with Lasagne
- Exercises – Basics of Theano
- Hands-on example – Build your own classifier from VGG-16

All the material is online at `github.com/mila-udem/summerschool2017`

Hands-on examples

Go to `http://mila.umontreal.ca/vmip`

- Jupyter notebooks
- Executed on AWS instances with a GPU (K80)

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

Current status

- Mature: developed and used since January 2008 (9 years old)
- Theano 0.9 released in March 2017
- Driven $>$ 1000 research papers
- Many contributors (123 for version 0.9)
- 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

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**

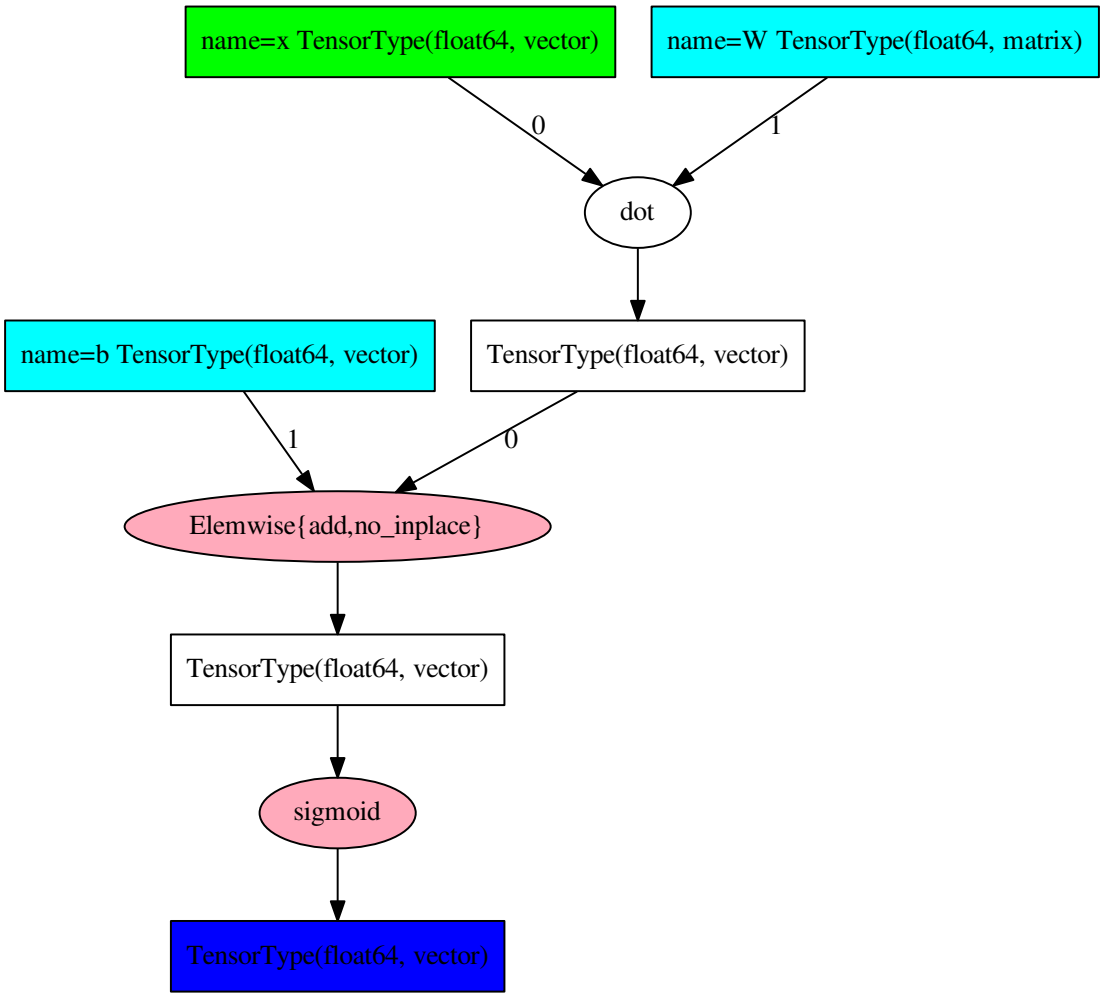
Build an expression

NumPy-like syntax

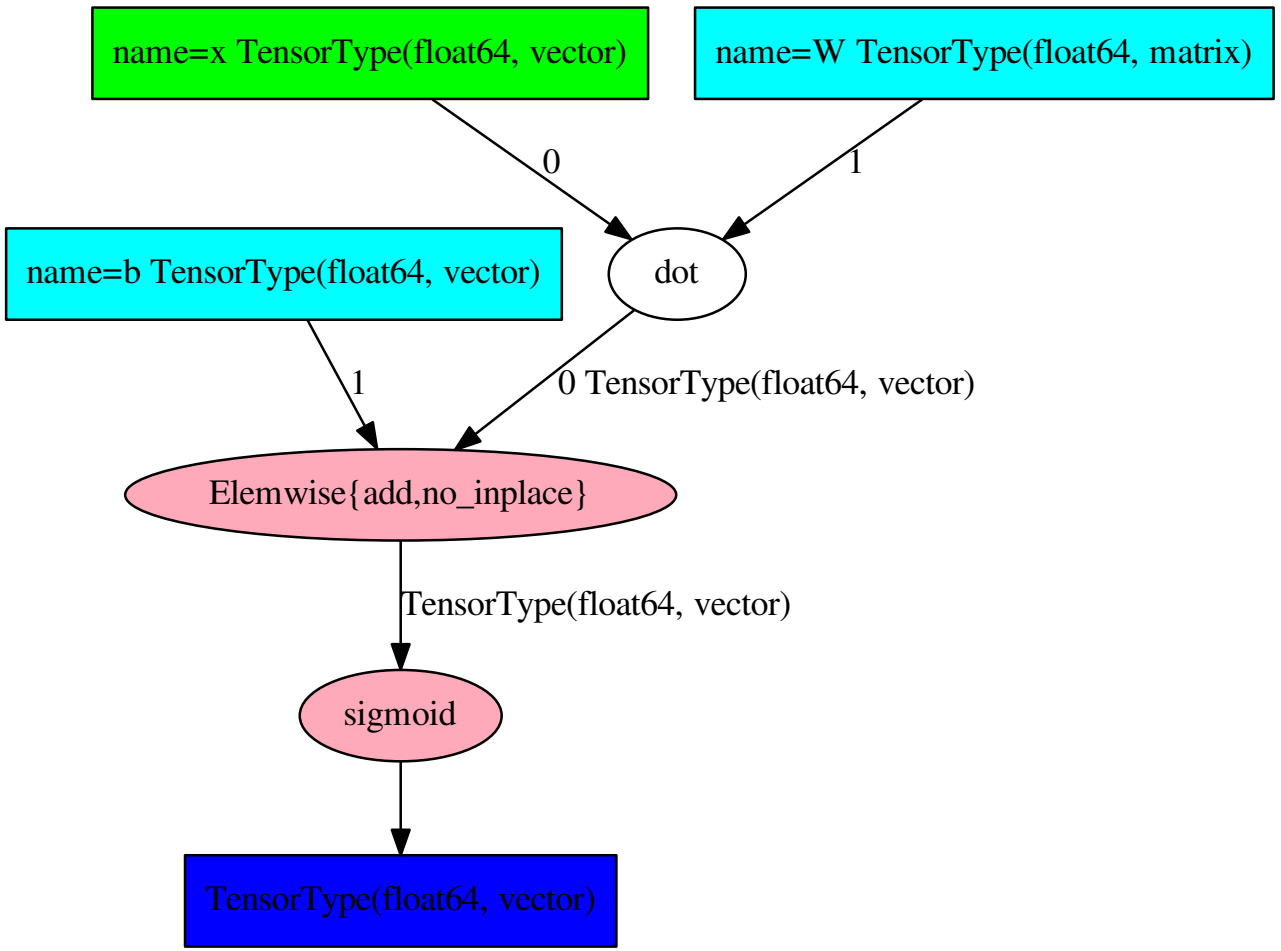
```
dot = T.dot(x, W)
out = T.nnet.sigmoid(dot + b)

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 $\mathbb{R}$.

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

The whole $M \times N$ Jacobian matrix $\frac{\partial g}{\partial x}\Big|_x$ is not needed.
We only need $\nabla g_x : \mathbb{R}^M \to \mathbb{R}^N, v \mapsto v \cdot \frac{\partial g}{\partial x}\Big|_x$
This is implemented for (almost) each mathematical operation in Theano.

Using theano.grad

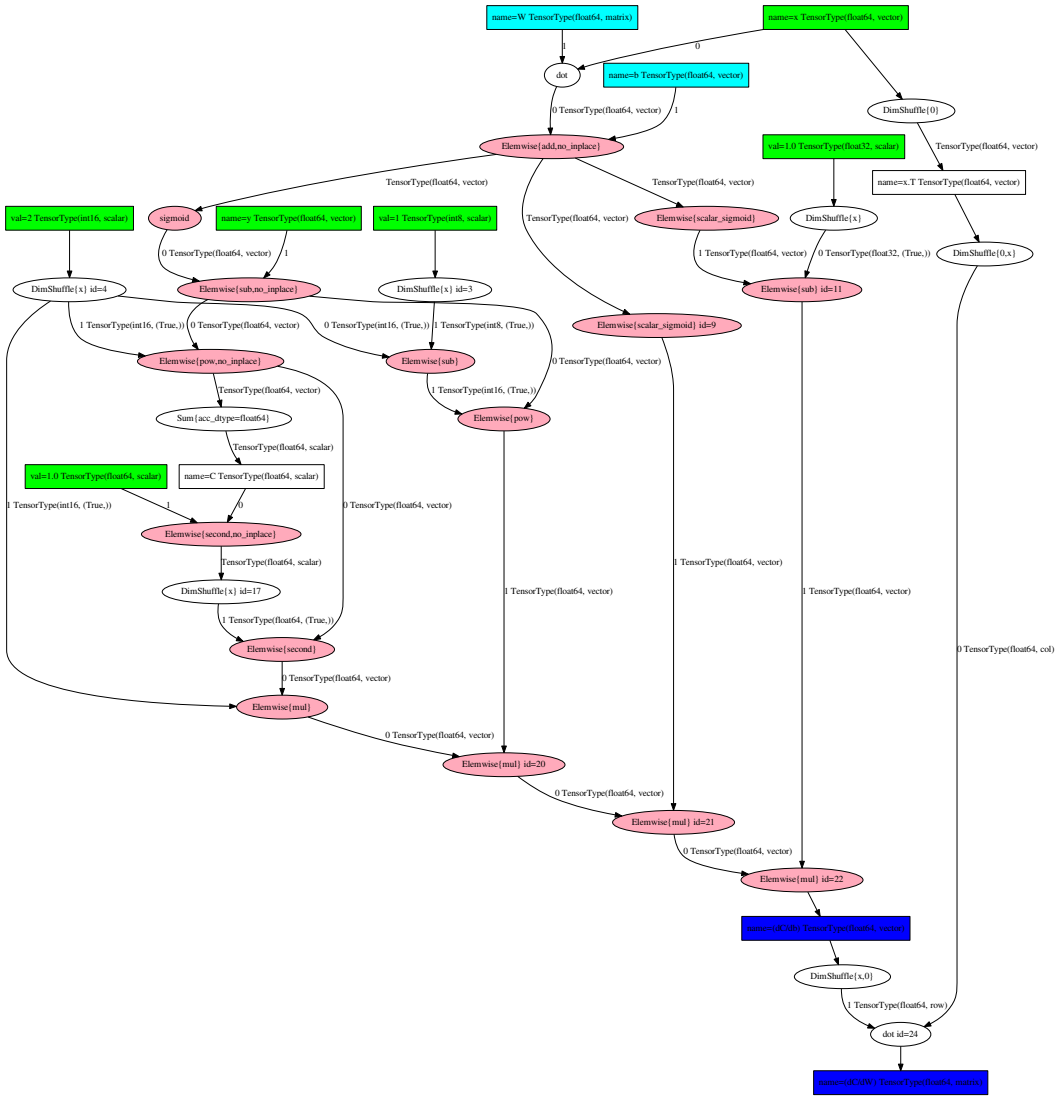
theano.grad traverses the graph, applying the chain rule.

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

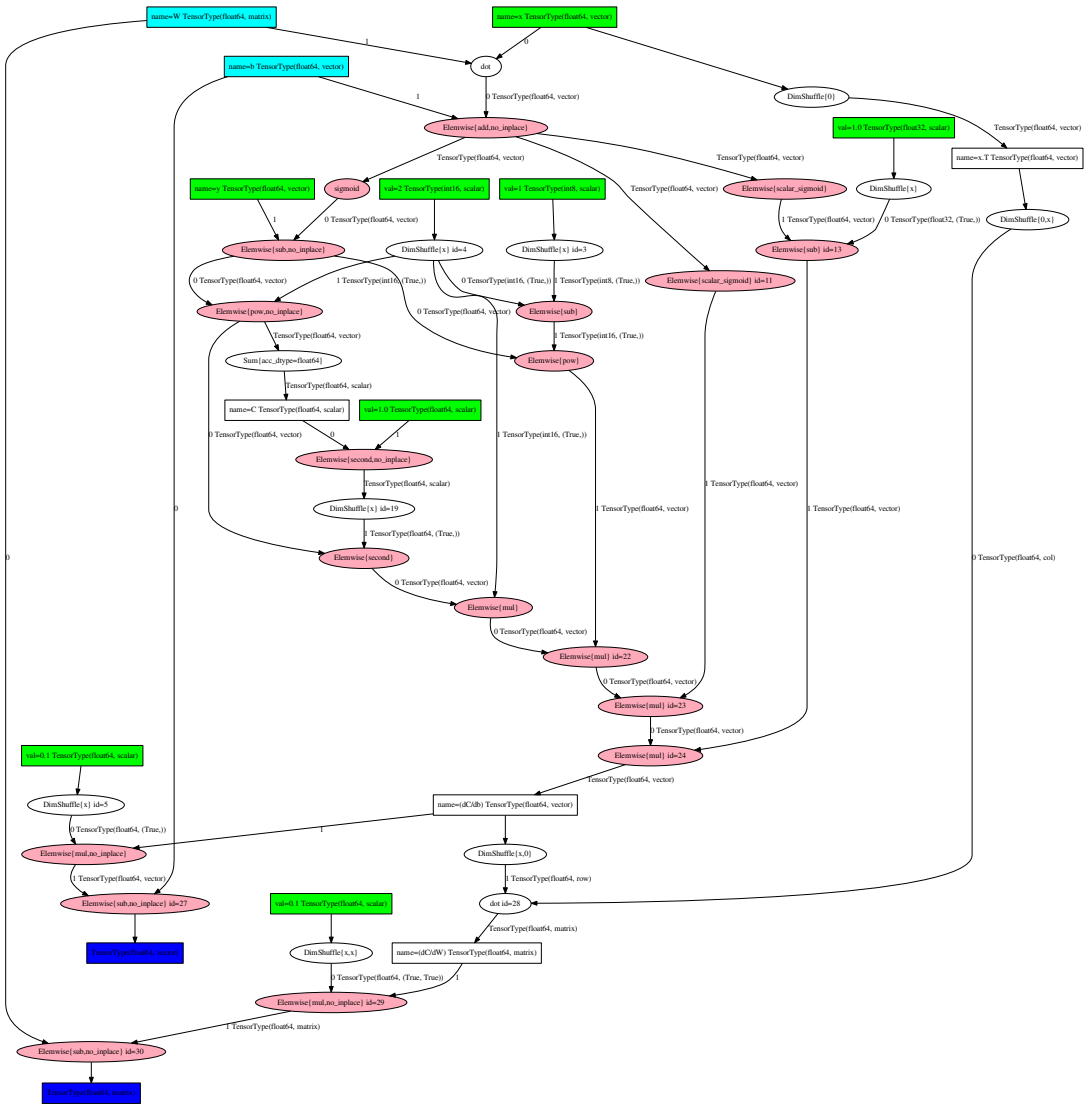
- dC_dW and dC_db are symbolic expressions, like out and C
- There are no numerical values at this point
- They are part of the same computation graph
- They can also be used to build new expressions

```
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:

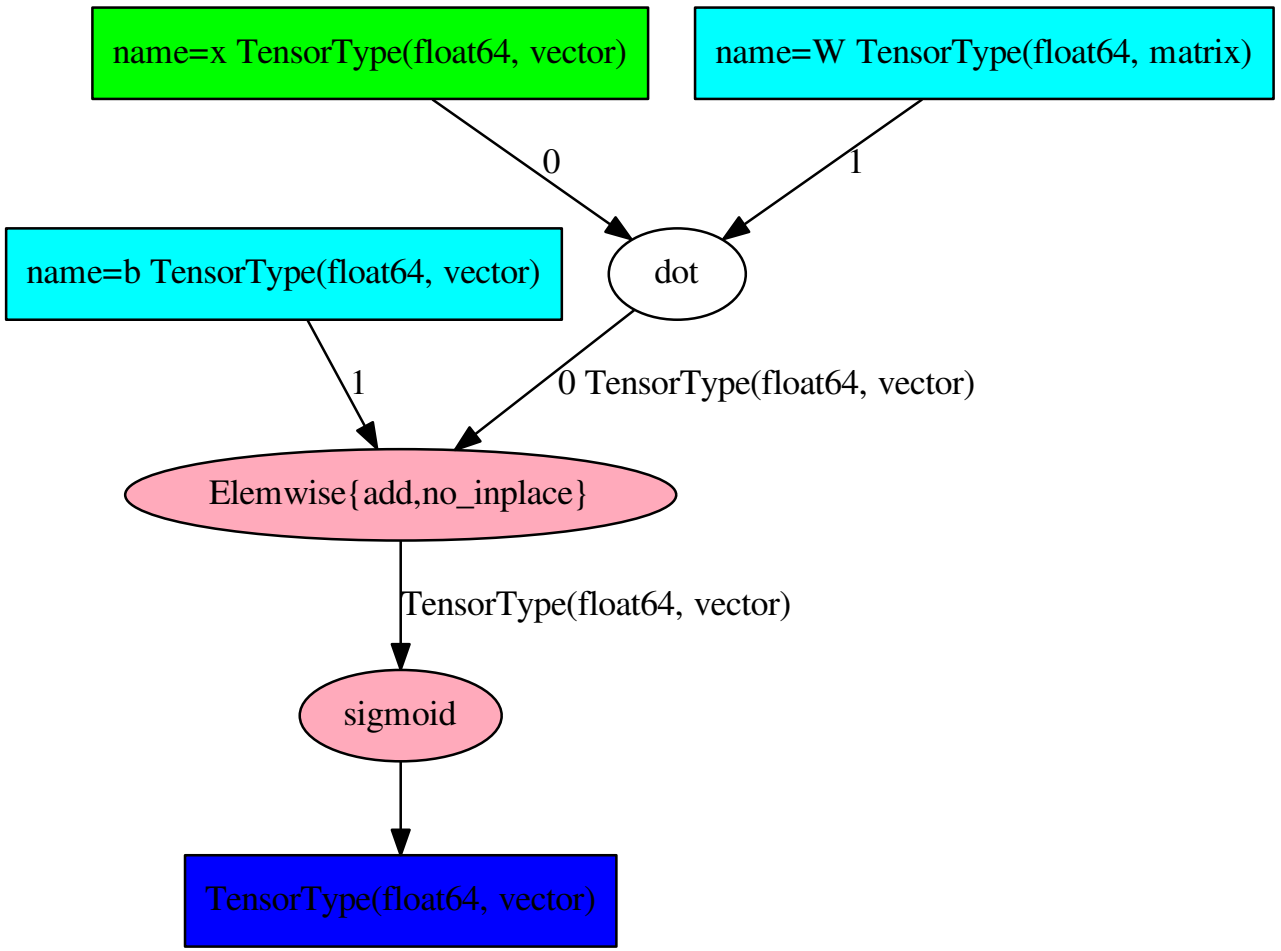
- Merge equivalent computations
- Simplify expressions: x/x becomes 1
- Numerical stability: "log$(1 + x)$" becomes "log1p(x)"
- Insert in-place an destructive versions of operations
- Use specialized, efficient versions (Elemwise loop fusion, BLAS, cuDNN)
- Shape inference
- Constant folding
- Transfer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

Enabling/disabling optimizations

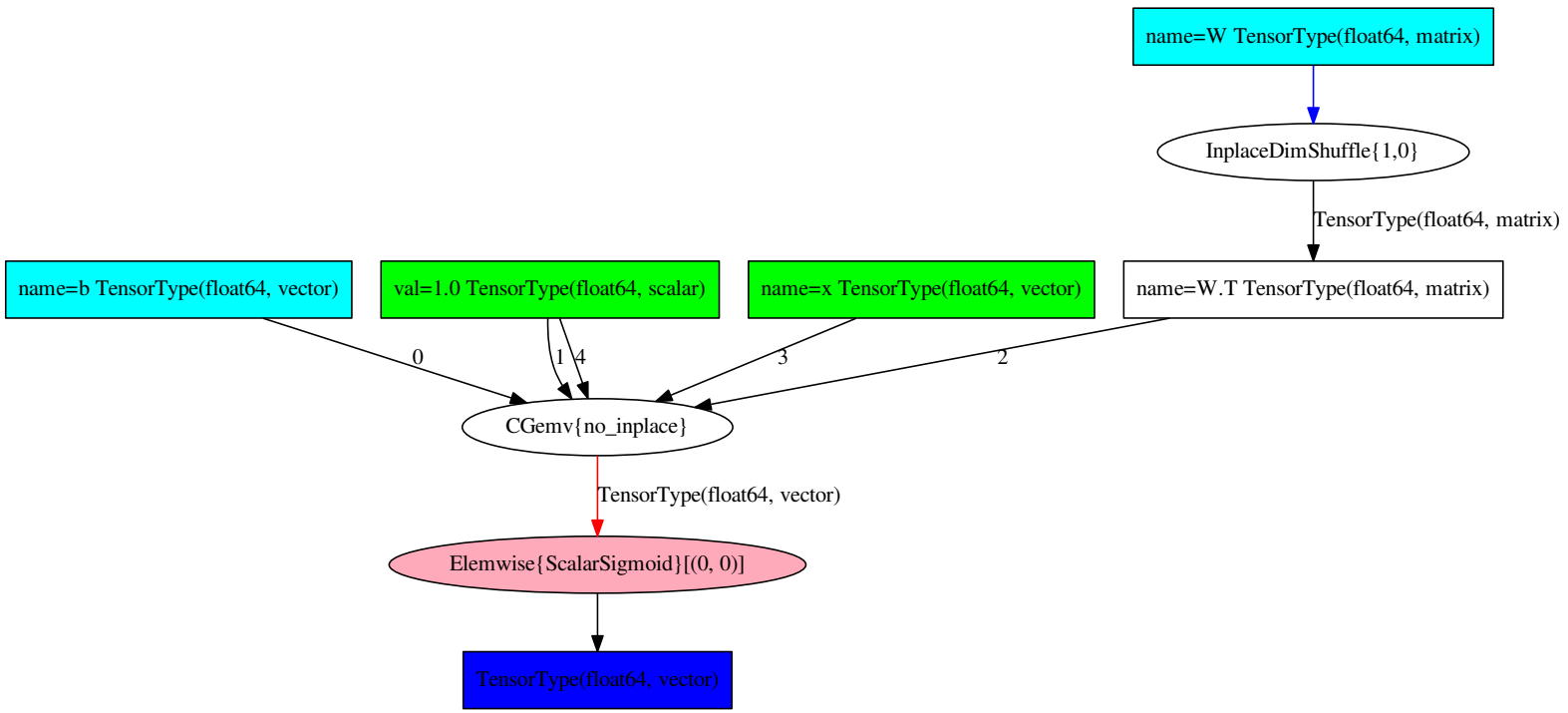
Trade-off between compilation speed, execution speed, error detection. Different pre-defined modes and optimizers govern the runtime and how much optimizations are applied

- `mode='FAST_RUN'`: default, make the runtime as fast as possible, launching overhead. Includes moving computation to GPU if a GPU was selected
- `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

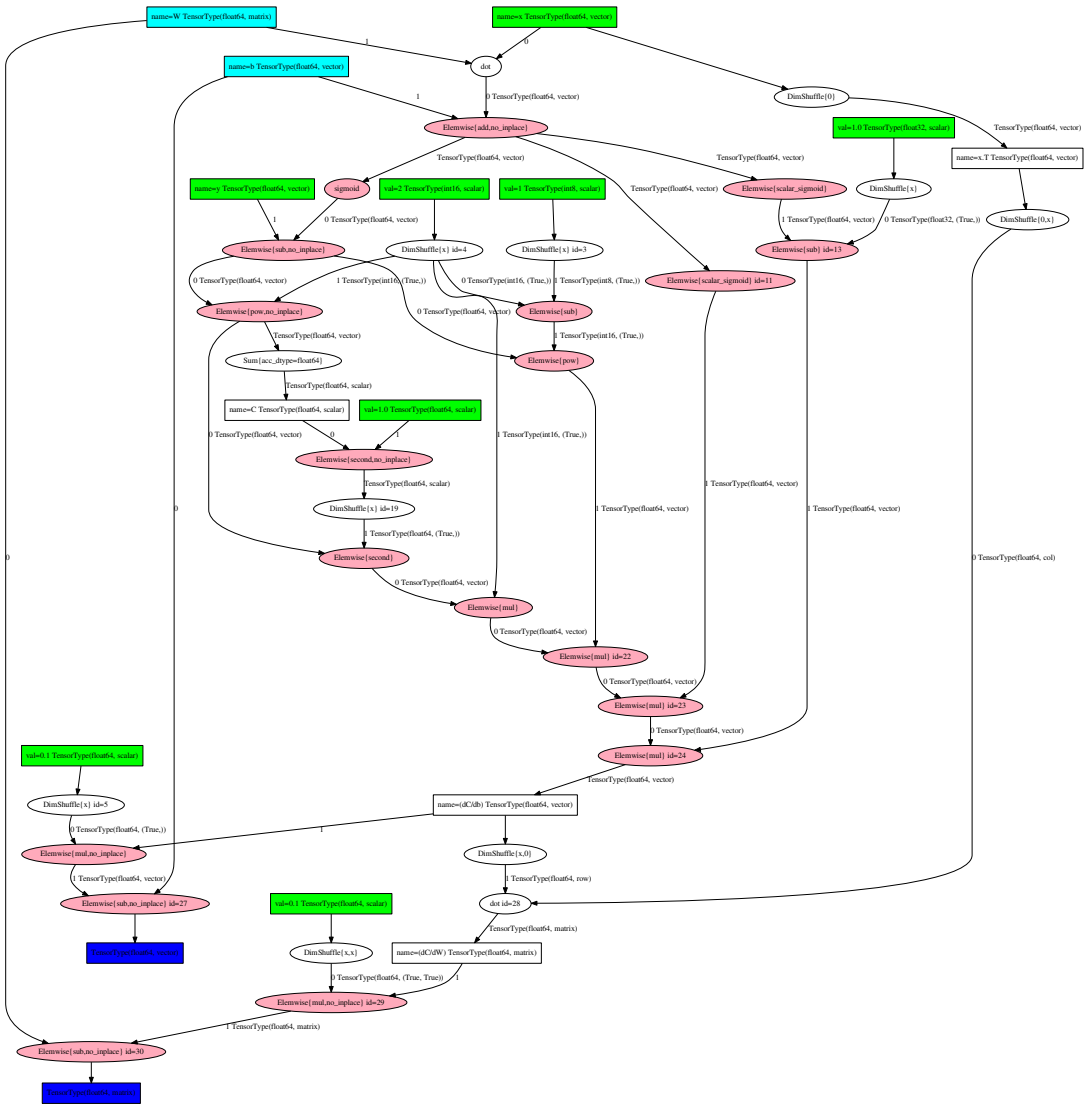
pydotprint(out)



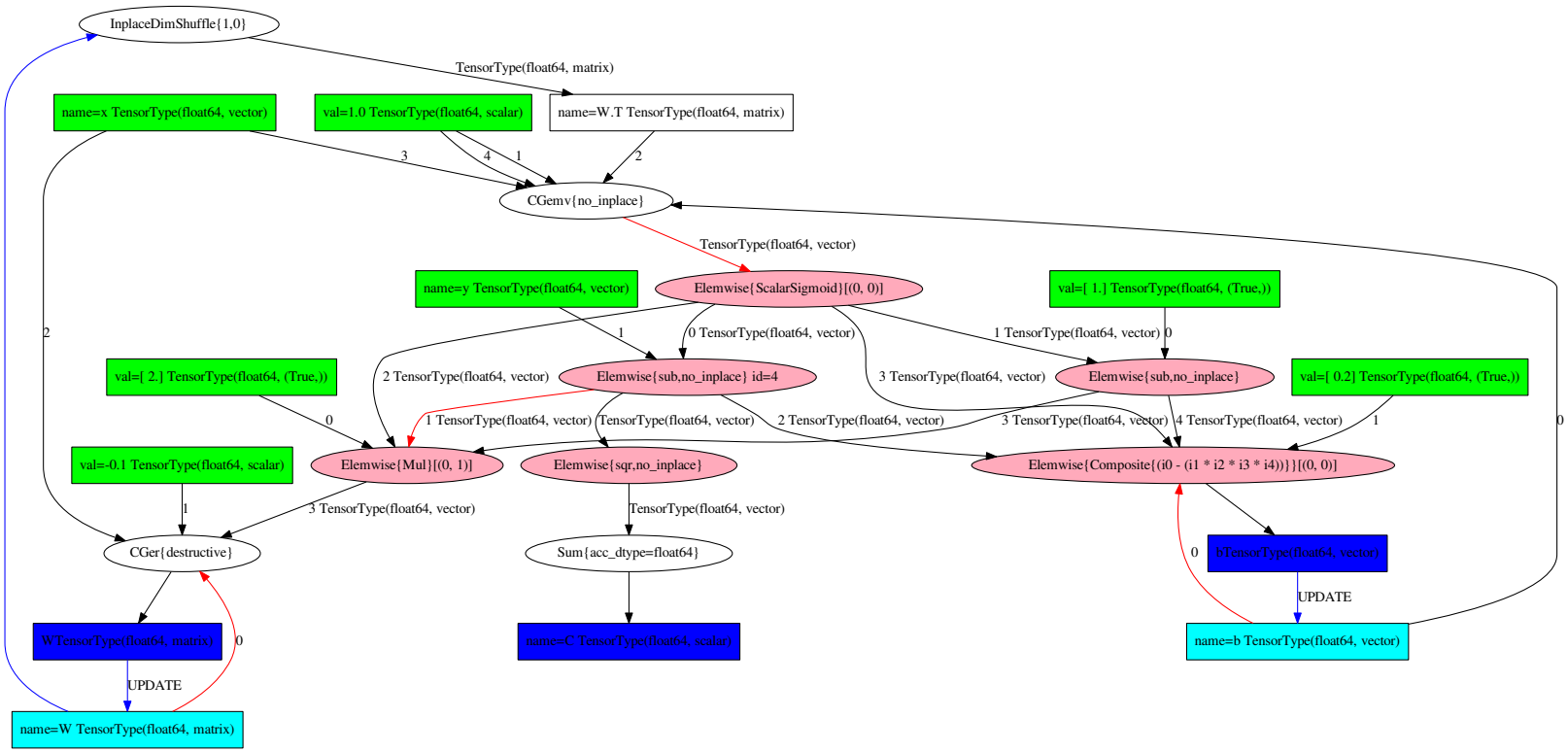
pydotprint(predict)



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
- Enables optimizations moving supported operations to GPU
- You want to make sure to use `float32` for speed

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')`)

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

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

Extending Theano

Theano can be extended in a few different ways

- Creating an Op with Python code
  - Easy, using Python bindings for specialized libraries (PyCUDA, . . .)
  - Some runtime overhead is possible
  - Example: 3D convolution using FFT on GPU
- Creating an Op with C or CUDA code
  - Use the C-API of Python / NumPy / GpuArray, manage refcounts
  - No overhead of Python function calls, or from the interpreter
  - C++ code inline or in a separate file
  - Example: Caffe-style convolutions, using GEMM, on CPU and GPU
- Adding an optimization
  - Perform additional graph simplifications
  - Replace part of the graph by a new optimized Op

New features

- New GPU back-end, based on `libgpuarray`, with:
  - Arrays of all dtypes, half-precision float (`float16`) for storage
  - Better scheduling
  - Much simpler installation on Windows (conda package)
- Performance improvements
  - Integration of CuDNN (now v6) for 2D/3D convolutions and pooling, RNNs, batch normalization
  - Fast memory allocator on GPU
  - For memory: checkpointing in scan, gradients of long sequences
  - Data parallelism with Platoon (`github.com/mila-udem/platoon/`)
- Faster graph optimization phase
  - More optimization / compile time trade-offs (`optimizer={o0,o1,...,o4}`)
  - Various ways to avoid recompilation
- Diagnostic tools
  - Interactive visualization (d3viz)
  - `PdbBreakPoint`

Current development

- Better support for int operations on GPU (indexing, argmax)
- Faster reductions on GPU
- Simpler, faster optimization mode
- Faster generation and loading of C++ / CUDA code
- More convolution variants: grouped, dilated, ... (GSoC)
- More linear algebra operations on GPU (GSoC)
- Data parallelism across nodes in Platoon
- `OpFromGraph` for re-defining gradients

Projects in our road map

- Constant shape inference when building the graph
- Better compilation cache for generated C++ code
- Continue refactoring graph optimization (for optimization speed)
- Optimize and re-use sub-graphs (like subroutines)
  - Improving `OpFromGraph`
  - Maybe cache them
- Deterministic mode
- Use CPU memory to offload intermediate results from GPU (maybe limited to Pascal GPUs)

What is Lasagne?

Lasagne is a thin framework/library on top of Theano.
`lasagne.readthedocs.org`

- Does not hide Theano
- Builds Theano graphs easily by using layers
- Contains many preimplemented losses and optimizers
- Does not include a training loop

Acknowledgements

- All people working or having worked at the MILA (previously LISA), especially Theano contributors
  - Reyhane Askari Hemmat, Frédéric Bastien, Yoshua Bengio, James Bergstra, Arnaud Bergeron, Steven Bocco, Philemon Brakel, Olivier Breuleux, Pierre Luc Carrier, Mathieu Germain, Ian Goodfellow, Simon Lefrançois, Razvan Pascanu, Joseph Turian, David Warde-Farley, and many more
- Compute Canada, Calcul Québec, NSERC, the Canada Research Chairs, CIFAR, and the CFI for providing funding or access to compute resources
- CIFAR and CRM for organizing the Deep learning summer school

Thanks for your attention

Questions, comments, requests?

Thanks for your attention

Questions, comments, requests?

`github.com/mila-udem/summerschool2017`

- Slides: `theano.pdf`
- Companion notebook: `notebooks/intro_theano.ipynb`

Thanks for your attention

Questions, comments, requests?

`github.com/mila-udem/summerschool2017`

- Slides: `theano.pdf`
- Companion notebook: `notebooks/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`
- Deep Learning Tutorials: `deeplearning.net/tutorial/`

Examples

Go to `http://mila.umontreal.ca/vmip`