**Theano at a Glance**

Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical.

**What does it do that they don’t?**

* **execution speed optimizations**: Theano can use g++ or nvcc to compile parts your expression graph into CPU or GPU instructions, which run much faster than pure Python.
* **symbolic differentiation**: Theano can automatically build symbolic graphs for computing gradients.
* **stability optimizations:** Theano can recognize [some] numerically unstable expressions and compute them with more stable algorithms.

The closest Python package to Theano is[**sympy**](http://code.google.com/p/sympy/). Theano focuses more on **tensor expressions** than Sympy, and has more machinery for compilation. Sympy has more sophisticated algebra rules and can handle a wider variety of mathematical operations (such as series, limits, and integrals).

**Theano Vision**  
Support tensor and sparse operations

* Support linear algebra operations
* Graph Transformations
  + Differentiation/higher order differentiation
  + ‘R’ and ‘L’ differential operators
  + Speed/memory optimizations
  + Numerical stability optimizations
* Can use many compiled languages, instructions sets: C/C++, CUDA, OpenCL, PTX, CAL, AVX, ...
* Lazy evaluation
* Loop
* Parallel execution (SIMD, multi-core, multi-node on cluster, multi-node distributed)
* Support all NumPy/basic SciPy functionality
* Easy wrapping of library functions in Theano

**Note: There is no short term plan to support multi-node computation.**

**Baby Steps – Algebra**

If you are following along and typing into an interpreter, you may have noticed that there was a slight delay in executing thefunctioninstruction. Behind the scene, **f was being compiled into C code.**

**Adding two Scalars**

Step 1

In Theano, all symbols must be typed. In particular, T.dscalar is the type we assign to “0-dimensional arrays (scalar) of doubles (d)”. It is a Theano [Type](http://deeplearning.net/software/theano/extending/graphstructures.html" \l "type).

Dscalar is not a class. Therefore, neither x nor y are actually instances of dscalar. They are instances of **TensorVariable**. X and y are, however, assigned the theano Type dscalar in their type field, as you can see here:

**>>>** x = T.dscalar('x')

**>>>** y = T.dscalar('y')

By calling T.dscalar with a string argument, you create a Variable representing a floating-point scalar quantity with the given name. If you provide no argument, the symbol will be unnamed. Names are not required, but they can help debugging.

Step 2

**>>>** z = x + y

z is yet another Variable which represents the addition of x and y. You can use the [pp](http://deeplearning.net/software/theano/library/printing.html" \l "libdoc-printing) function to pretty-print out the computation associated to z.

Step 3

The last step is to create a function taking x and y as inputs and giving z as output:

f = function([x, y], z)

The first argument to **[function](http://deeplearning.net/software/theano/library/compile/function.html" \l "function.function)** is a list of Variables that will be provided as inputs to the function. The second argument is a single Variable or a list of Variables. For either case, the second argument is what we want to see as output when we apply the function. f may then be used like a normal Python function.

Computing More than one Thing at the Same Time

**>>>** a, b = T.dmatrices('a', 'b')

**>>>** diff = a - b

**>>>** abs\_diff = abs(diff)

**>>>** diff\_squared = diff\*\*2

**>>>** f = function([a, b], [diff, abs\_diff, diff\_squared])

## Setting a Default Value for an Argument

**>>> from** **theano** **import** Param

**>>>** x, y = T.dscalars('x', 'y')

**>>>** z = x + y

**>>>** f = function([x, Param(y, default=1)], z)

**>>>** f(33)

array(34.0)

**>>>** f(33, 2)

array(35.0)

## Using Shared Variables

**>>> from** **theano** **import** shared

**>>>** state = shared(0)

**>>>** inc = T.iscalar('inc')

**>>>** accumulator = function([inc], state, updates=[(state, state+inc)])

This code introduces a few new concepts. The shared function constructs so-called [shared variables](http://deeplearning.net/software/theano/library/compile/shared.html" \l "libdoc-compile-shared). These are hybrid symbolic and non-symbolic variables whose value may be shared between multiple functions.

The other new thing in this code is the updates parameter of function. updates must be supplied with a list of pairs of the form (shared-variable, new expression). It can also be a dictionary whose keys are shared-variables and values are the new expressions.

**>>>** state.get\_value()

array(0)

**>>>** accumulator(1)

array(0)

**>>>** state.get\_value()

array(1)

**>>>** accumulator(300)

array(1)

**>>>** state.get\_value()

array(301)

**>>>** state.set\_value(-1)

**>>>** accumulator(3)

array(-1)

**>>>** state.get\_value()

array(2)

The updates mechanism can be a syntactic convenience, but it is mainly there for efficiency. Updates to shared variables can sometimes be done more quickly using in-place algorithms (e.g. low-rank matrix updates). Also, Theano has more control over where and how shared variables are allocated, which is one of the important elements of getting good performance on the [GPU](http://deeplearning.net/software/theano/tutorial/using_gpu.html" \l "using-gpu).

It may happen that you expressed some formula using a shared variable, but you do not want to use its value. In this case, you can use the givens parameter of function which replaces a particular node in a graph for the purpose of one particular function.

**>>>** fn\_of\_state = state \* 2 + inc

**>>>** *# The type of foo must match the shared variable we are replacing*

**>>>** *# with the ``givens``*

**>>>** foo = T.scalar(dtype=state.dtype)

**>>>** skip\_shared = function([inc, foo], fn\_of\_state,

givens=[(state, foo)])

**>>>** skip\_shared(1, 3) *# we're using 3 for the state, not state.value*

array(7)

**>>>** state.get\_value() *# old state still there, but we didn't use it*

array(0)

## **Theano Graphs**

The first step in writing Theano code is to write down all mathematical relations using symbolic placeholders (variables). When writing down these expressions you use operations like +, -, \*\*, sum(), tanh(). All these are represented internally as ops. An op represents a certain computation on some type of inputs producing some type of output. You can see it as afunction definition in most programming languages.

**Graph Structures**

**Apply**

An Apply nodeis a type of internal node used to represent a [computation graph](http://deeplearning.net/software/theano/glossary.html" \l "term-graph)in Theano. Unlike [Variable nodes](http://deeplearning.net/software/theano/extending/graphstructures.html" \l "variable), Apply nodes are usually not manipulated directly by the end user. They may be accessed via a Variable’s ownerfield.

**Op**

An [Op](http://deeplearning.net/software/theano/extending/graphstructures.html" \l "op)in Theano defines a certain computation on some types of inputs, producing some types of outputs. It is equivalent to a function definition in most programming languages. From a list of input [Variables](http://deeplearning.net/software/theano/extending/graphstructures.html" \l "variable)and an Op, you can build an [Apply](http://deeplearning.net/software/theano/extending/graphstructures.html" \l "apply)node representing the application of the Op to the inputs.

It is important to understand the distinction between an Op (the definition of a function) and an Apply node (the application of a function). If you were to interpret the Python language using Theano’s structures, code going like def f(x): ... would produce an Op for f whereas code like a = f(x) or g(f(4), 5) would produce an Apply node involving the f Op.