Theano Tutorials
Session 2
Advanced usage

Ian Goodfellow

Outline

- Manipulating symbolic expressions
- Debugging

Manipulating symbolic 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>
```

Holding variables constant

- Directly in the call to grad:
  - consider_constant allows you to specify a list of variables to hold constant
  - flexible, but easy to forget
- block_gradient
  - Not in Theano, in pylearn2.utils
  - Makes a constant out of any expression
  - Less flexible, but no risk of forgetting

Combining gradients from other sources

- Suppose z(x) = f(g(x))
- Suppose we want to split backprop into two steps:
  - backprop through f
  - backprop through g
- This can be more many reasons:
  - Maybe f is not a Theano function
  - Saving memory

known_grads

- Parameter of the grad method
- Specify a dictionary mapping variables to gradients that are already known
- Don't even need to specify a cost if a separating set is already known

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

Exercises

- Work through the “01_symbolic” directory now

Debugging

- DEBUG_MODE
- compute_test_value
- min_informative_str
- DebugPrint
- Print
- Accessing the FunctionGraph
- WrapLinkers

compute_test_value

```
>>> from theano import config
>>> config.compute_test_value = 'raise'
>>> x = T.vector()
>>> import numpy as np
>>> x.tag.test_value = np.ones((2,))
>>> y = T.vector()
>>> y.tag.test_value = np.ones((3,))
>>> x + y
...
ValueError: Input dimension mis-match.
(input[0].shape[0] = 2, input[1].shape[0] = 3)
```

min_informative_str

```
>>> x = T.scalar()
>>> y = T.scalar()
>>> z = x + y
>>> z.name = 'z'
>>> a = 2. * z
>>> from theano.printing import min_informative_str
>>> print min_informative_str(a)
A. Elemwise{mul,no_inplace}
 B. TensorConstant{2.0}
 C. z
```

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]
```

Print

```
x = theano.tensor.vector()
x = theano.printing.Print('x', attrs=['min','max'])(x)
```

Accessing a function's fgraph

```
>>> x = T.scalar()
>>> y = x / x
>>> f = function([x], y)
>>> debugprint(f.maker.fgraph.outputs[0])
DeepCopyOp [@A] "
 |TensorConstant{1.0} [@B]
```

WrapLinkers

- In theano terms, a *Linker* is an object that drives the execution of the function. Goes through all Apply nodes and calls the Op's code on the inputs to generate the outputs
- You can write your own linker to wrap each of these individual calls with extra functionality
- Example uses:
  - Test that behavior is deterministic by having one Wraplinker that saves everything and one that reloads it and checks that the new values match.
  - Raise an error if any value is ever NaN

WrapLinkers continued

```
from theano.compile import Mode

def my_callback(i, node, fn):
   # add any preprocessing here
   fn()
   # add any postprocessing here

class MyWrapLinker(Mode):
   def __init__(self):
      wrap_linker = theano.gof.WrapLinkerMany(
                  [theano.gof.OpWiseCLinker()],
                  [my_callback])
      super(MyWrapLinker, self).__init__(wrap_linker,
                  optimizer='fast_run')

my_mode = MyWrapLinker()
f = function(inputs, outputs, mode=my_mode)
```

Exercises

- Work through the “02_debugging” directory now