Extending Theano

Arnaud Bergeron

October 29, 2014

Outline

1. How to Make an Op (Python) (45 min)
2. How to Make an Op (C) (30 min)
3. How to Make a Complex Op (10 min)
4. Optimizations (20 min)

How to Make an Op (Python)

Overview

```
from theano import Op

class MyOp(Op):
    __props__ = ()

    def __init__(self, ...):
        # set up parameters

    def make_node(self, ...):
        # create apply node

    def perform(self, node, inputs, outputs_storage):
        # do the computation
```

__init__

```
    def __init__(self, ...):
        # set up parameters
```

- Optional, a lot of Ops don't have one
- Serves to set up Op-level parameters
- Should also perform validation on those parameters

__props__

```
    __props__ = ()
```

- Optional (although very useful)
- Generates `__hash__`, `__eq__` and `__str__` methods if present
- Empty tuple signifies no properties that should take part in comparison
- If you have only one property, make sure you add a final comma: `('property',)`

Make sure `__hash__`, `__eq__` and `__str__` are not defined in a superclass if you don't inherit directly from Op since otherwise your methods will get shadowed.

make_node

```
    def make_node(self, ...):
        # create apply node
```

- This creates the node object that represents our computation in the graph
- The parameters are usually Theano variables, but can be python objects too
- The return value must be an `Apply` instance

perform

```
def perform(self, node, inputs, outputs_storage):
    # do the computation
```

- This performs the computation on a set of values (hence the method name)
- The parameters are all python objects (not symbolic values)
- This method must not return its result, but rather store it in the 1-element lists (or cells) provided in `output_storage`

DoubleOp

```
from theano import Op, Apply
from theano.tensor import as_tensor_variable

class DoubleOp(Op):
    __props__ = ()

    def make_node(self, x):
        x = as_tensor_variable(x)
        return Apply(self, [x], [x.type()])

    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = x * 2
```

Op Instances and Nodes

When you call an op class you get an instance of that Op:

```
double_op = DoubleOp()
```

But when you want to use that op as a node in a graph you need to call the *instance*:

```
node = double_op(x)
```

You can do both steps at once with a double call like this:

```
node = DoubleOp()(x)
```

Basic Tests

```
import numpy

from theano import function, config
from theano.tensor import matrix

from doubleop import DoubleOp

def test_doubleop():
    x = matrix()
    f = function([x], DoubleOp()(x))
    inp = numpy.asarray(numpy.random.rand(5, 4),
                        dtype=config.floatX)
    out = f(inp)
    utt.assert_allclose(inp * 2, out)
```

Run Tests

The simplest way to run your tests is to use `nosetests` directly on your test file like this:

```
$ nosetests test_doubleop.py
.

----------------------------------------------------------------------
Ran 1 test in 0.427s

OK
```

You can also use `theano-nose` which is a wrapper around `nosetests` with some extra options.

Exercise: TripleOp

What would need to be changed in the code below (DoubleOp) to make this Op triple the input instead of double?

```
from theano import Op, Apply
from theano.tensor import as_tensor_variable

class DoubleOp(Op):
    __props__ = ()

    def make_node(self, x):
        x = as_tensor_variable(x)
        return Apply(self, [x], [x.type()])

    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = x * 2
```

Solution: TripleOp

You change the class name and the constant `2` for a constant `3`.

```
from theano import Op, Apply
from theano.tensor import as_tensor_variable

class TripleOp(Op):
    __props__ = ()

    def make_node(self, x):
        x = as_tensor_variable(x)
        return Apply(self, [x], [x.type()])

    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = x * 3
```

Exercise: ScalMulOp

Work though the "06_scalmulop" directory available at `https://github.com/abergeron/ccw_tutorial_theano.git`.

- Take the `DoubleOp` code and make it work with an arbitrary scalar
- There are more than one solution possible, both have advantages and disadvantages

infer_shape

```
def infer_shape(self, input_shapes):
    # return output shapes
```

- This functions is optional, although highly recommended
- It takes as input the symbolic shapes of the input variables
- `input_shapes` is of the form
  `[[i0_shp0, i0_shp1, ...], ...]`
- It must return a list with the symbolic shape of the output variables

Example

```
    def infer_shape(self, node, input_shapes):
        return input_shapes
```

- Here the code is really simple since we don't change the shape in any way in our Op
- `input_shapes` would be an expression equivalent to `[x.shape]`

Tests

To test the `infer_shape` method we use `InferShapeTester`

```
from theano.tests import unittest_tools as utt

class test_Double(utt.InferShapeTester):
    def test_infer_shape(self):
        x = matrix()
        self._compile_and_check(
            # function inputs (symbolic)
            [x],
            # Op instance
            [DoubleOp()(x)],
            # numeric input
            [numpy.asarray(numpy.random.rand(5, 4),
                           dtype=config.floatX)],
            # Op class that should disappear
            DoubleOp)
```

Gradient

```
def grad(self, inputs, output_grads):
    # return gradient graph for each input
```

- This function is required for graphs including your op to work with `theano.grad()`
- It must return a list of symbolic graphs for each of your inputs
- You compute the partial derivative with regards to your inputs
- Inputs that have no valid gradient should have a special `DisconnectedType` value

Example

```
    def grad(self, inputs, output_grads):
        return [output_grads[0] * 2]
```

- Here since the operation is simple the gradient is simple
- Note that we return a list

Tests

To test the gradient we use `verify_grad`

```
from theano.tests import unittest_tools as utt

def test_doubleop_grad():
    utt.verify_grad(
        # Op instance
        DoubleOp(),
        # Numeric inputs
        [numpy.random.rand(5, 7, 2)]
        )
```

It will compute the gradient numerically and symbolically (using our `grad()` method) and compare the two.

Exercice: Add Special Methods to ScalMulOp

Work through the "07_scalmulgrad" directory available at `https://github.com/abergeron/ccw_tutorial_theano.git`

- Take the ScalMulOp class you made and add the `infer_shape` and `grad` methods to it.
- Don't forget to make tests for your new class to make sure everything works correctly.

How to Make an Op (C)

Overview

```
from theano import Op

class MyOp(Op):
    __props__ = ()

    def make_node(self, ...):
        # return apply node

    def c_code(self, node, name, input_names,
               output_names, sub):
        # return C code string

    def c_support_code(self):
        # return C code string

    def c_code_cache_version(self):
        # return hashable object
```

c_code

```
def c_code(self, node, name, input_names,
           output_names, sub):
    # return C code string
```

- This method returns a python string containing C code
- `input_names` contains the variable names where the inputs are
- `output_names` contains the variable names where to place the outputs
- `sub` contains some code snippets to insert into our code (mostly to indicate failure)

Support Code

```
def c_support_code(self):
    # return C code string
```

- This method return a python string containing C code
- The code may be shared with multiple instances of the op
- It can contain things like helper functions

There are a number of similar methods to insert code at various points

Headers, Libraries, Compilers

Some of the methods available to customize the compilation environment:

`c_libraries` Return a list of shared libraries the op needs

`c_headers` Return a list of included headers the op needs

`c_compiler` C compiler to use (if not the default)

Again others are available. Refer to the documentation for a complete list.

Example I

This is the C code equivalent to `perform`

```
from theano import Op, Apply
from theano.tensor import as_tensor_variable

class DoubleC(Op):
    __props__ = ()

    def make_node(self, x):
        x = as_tensor_variable(x)
        if x.ndim != 1:
            raise TypeError("DoubleC only works on 1D")
        return Apply(self, [x], [x.type()])
```

Example II

```
    def c_code(self, node, name, input_names,
               output_names, sub):
        return """
Py_XDECREF(%(out)s);
%(out)s = (PyArrayObject *)PyArray_NewLikeArray(
    %(inp)s, NPY_ANYORDER, NULL, 0);
if (%(out)s == NULL) {
  %(fail)s
}
for (npy_intp i = 0; i < PyArray_DIM(%(inp)s, 0); i++) {
  *(dtype_%(out)s *)PyArray_GETPTR1(%(out)s, i) =
    (*(dtype_%(inp)s *)PyArray_GETPTR1(%(inp)s, i)) * 2;
}
""" % dict(inp=input_names[0], out=output_names[0],
           fail=sub["fail"])
```

COp

```
from theano.gof import COp

class MyOp(COp):
    __props__ = ()

    def __init__(self, ...):
        COp.__init__(self, c_file, func_name)
        # Other init code if needed

    def make_node(self, ...):
        # make the Apply node
```

Constructor Arguments

- Basically you just pass two arguments to the constructor of COp
  - Either by calling the constructor directly
    `COp.__init__(self, ...)`
  - Or via the superclass `super(MyOp, self).__init__(...)`
- The two arguments are:
  - the name of the C code file
  - the name of the function to call to make the computation

COp: Example

```
from theano import Apply
from theano.gof import COp
from theano.tensor import as_tensor_variable

class DoubleCOp(COp):
    __props__ = ()

    def __init__(self):
        COp.__init__(self, "./doublecop.c",
                     "APPLY_SPECIFIC(doublecop)")

    def make_node(self, x):
        x = as_tensor_variable(x)
        if x.ndim != 1:
            raise TypeError("DoubleCOp only works with 1D")
        return Apply(self, [x], [x.type()])
```

COp: Example

```
THEANO_APPLY_CODE_SECTION

int APPLY_SPECIFIC(doublecop)(PyArrayObject *x,
                              PyArrayObject **out) {
  Py_XDECREF(*out);
  *out = (PyArrayObject *)PyArray_NewLikeArray(
                            inp, NPY_ANYORDER, NULL, 0);
  if (*out == NULL)
    return -1;

  for (npy_intp i = 0; i < PyArray_DIM(x, 0); i++) {
    *(DTYPE_OUTPUT_0 *)PyArray_GETPTR1(*out, i) =
      (*(DTYPE_INPUT_0 *)PyArray_GETPTR1(x, i)) * 2;
  }
  return 0;
}
```

Tests

- Testing ops with C code is done the same way as testing for python ops
- One thing to watch for is tests for ops which don't have python code
  - You should skip the test in those cases
  - Test for `theano.config.gxx == ""`
- Using DebugMode will compare the output of the Python version to the output of the C version and raise an error if they don't match

Gradient and Other Concerns

- The code for `grad()` and `infer_shape()` is done the same way as for a python Op
- In fact you can have the same Op with a python and a C version sharing the `grad()` and `infer_shape()` code
  - That's how most Ops are implemented

Exercice: Add C Code to ScalMulOp

Work through the "08_scalmulc" directory available at `https://github.com/abergeron/ccw_tutorial_theano.git`.

- Take the ScalMulOp from before and write C code for it using either approach (only accept vectors).
- You can base yourself on the C code for DoubleOp.
- Don't forget to test your new implementation! Be sure to check for invalid inputs (matrices).

How to Make a Complex Op

make_thunk

```
    def make_thunk(self, node, storage_map,
                   compute_map, no_recycling):
        # return a thunk
```

- Define instead of `perform` or `c_code`
- Gives total freedom on how the computation is performed
- More complex to use and generally not needed

Optimizations

Purpose

- End goal is to make code run faster
- Sometimes they look after stability or memory usage
- Most of the time you will make one to insert a new Op you wrote

Replace an Op (V1)

Here is code to use `DoubleOp()` instead of `ScalMul(2)`.

```
from scalmulop import ScalMulV1
from doubleop import DoubleOp

from theano.gof import local_optimizer
@local_optimizer([ScalMulV1])
def local_scalmul_double_v1(node):
    if not (isinstance(node.op, ScalMulV1)
            and node.op.scal == 2):
        return False

    return [DoubleOp()(node.inputs[0])]
```

Replace an Op (V2)

In this case since we are replacing one instance with another there is an easier way.

```
from scalmulop import ScalMulV1
from doubleop import DoubleOp

from theano.gof.opt import OpSub

local_scalmul_double_v2 = OpSub(ScalMulV1(2), DoubleOp())
```

Registering

In any case you need to register your optimization.

```
from theano.tensor.opt import register_specialize

@register_specialize
@local_optimizer([ScalMulV1])
def local_scalmul_double_v1(node):
```

```
register_specialize(local_scalmul_double_v2,
                    name='local_scalmul_double_v2')
```

Tests

```
import theano

from scalmulop import ScalMulV1
from doubleop import DoubleOp
import opt

def test_scalmul_double():
    x = theano.tensor.matrix()
    y = ScalMulV1(2)(x)
    f = theano.function([x], y)

    assert not any(isinstance(n.op, ScalMulV1)
                   for n in f.maker.fgraph.toposort())
    assert any(isinstance(n.op, DoubleOp)
               for n in f.maker.fgraph.toposort())
```

Exercice 4

Work through the "09_opt" directory available at `https://github.com/abergeron/ccw_tutorial_theano.git`.

- Make an optimization that replace DoubleOp with DoubleC (or DoubleCOp)
- Write tests to make sure your optimization is applied correctly