Extending Theano

Arnaud Bergeron

February 1, 2017

Outline

1. How to Make an Op (Python) (45 min)
2. How to Make an Op (C) (30 min)
3. Op Params (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__ = ()
```

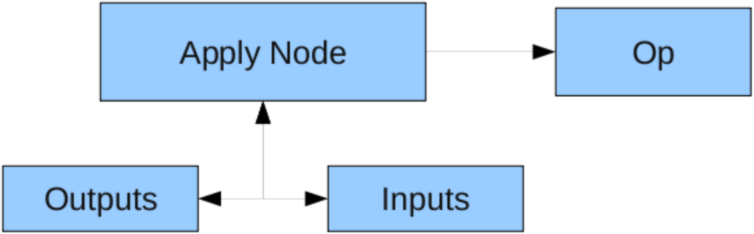
- Optional (although very useful)
- Generates `__hash__`, `__eq__` and `__str__` methods if present
- Empty tuple signifies no properties that should take part in comparison
- If you have only one property, make sure you add a final comma: `('property',)`

make_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

What Is an Apply Node?



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 `outputs_storage`
- The output storage may contain a pre-existing value from a previous run that may be reused for 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 theano.tests import unittest_tools as utt
from doubleop import DoubleOp


def test_doubleop():
    utt.seed_rng()
    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.

infer_shape

```
def infer_shape(self, node, 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

```
from theano.tests import unittest_tools as utt

class test_Double(utt.InferShapeTester):
    def test_infer_shape(self):
        utt.seed_rng()
        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

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

- This function is required for graphs including your op to work with `theano.grad()`
- Each item you return represents the gradient with respect to that input computed based on the gradient with respect to the outputs (which you get in `output_grads`).
- It must return a list of symbolic graphs for each of your inputs
- Inputs that have no valid gradient should have a special `DisconnectedType` value

Example

```
def L_op(self, inputs, outputs, 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.seed_rng()
    utt.verify_grad(
        # Op instance
        DoubleOp(),
        # Numeric inputs
        [numpy.random.rand(5, 7, 2)]
        )
```

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

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)
- The variables in `output_names` may contain a reference to a pre-existing value from a previous run that may be reused for storage.

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.

Python C-API

`void Py_INCREF(PyObject *o)` Increase the reference count of a python object.

`void Py_DECREF(PyObject *o)` Decrease the reference count of a python object.

`void Py_XINCREF(PyObject *o)` Increase the reference count of a (potentially NULL) python object.

`void Py_XDECREF(PyObject *o)` Decrease the reference count of a (potentially NULL) python object.

Numpy C-API

`int PyArray_NDIM(PyArrayObject *a)` Get the number of dimension of an array.

`npy_intp *PyArray_DIMS(PyArrayObject *a)` Get the shape of an array.

`npy_intp *PyArray_STRIDES(PyArrayObject *a)` Get the strides of an array.

`void * PyArray_DATA(PyArrayObject *a)` Get the data pointer (pointer to element 0) of an array.

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_files, func_name)
        # Other init code if needed

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

Constructor Arguments

- Basically you just pass arguments to the constructor of COp
  - Either by calling the constructor directly `COp.__init__(self, ...)`
  - Or via the superclass `super(MyOp, self).__init__(...)`
- The arguments are:
  - a list of file names with code sections (relative to the location of the op class)
  - the name of a function to call to make the computation (optional)

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

```c
#section support_code

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

Op Params

Purpose

- Used to pass information to the C code
- Can reduce the amount of compiled C code
- Required for things that can change from one script run to the other.

Usage

```
from theano import Op

class MyOp(Op):
    params_type = # a params type here

    def __init__(self, ...):
        # Get some params

    # signature change
    def perform(self, node, inputs, out_storage, params):
        # do something

    def get_params(self, node):
        # Return a params object
```

GPU Ops

Overview

```
from theano import Op
from theano.gpuarray.type import gpu_context_type

class GpuOp(Op):
    __props__ = ()
    params_type = gpu_context_type

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

    def get_params(self, node):
        return node.outputs[0].type.context
```

Overview

```
def perform(self, node, inputs, output_storage):
    # python code

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

- params_type is new.
- get_params is new.

Context and Context Name

- Context is what is used to refer to the chosen GPU.
  It is a C object that can't be serialized.
- Context Name is a name internal to Theano to refer to a given context object. It is a python string.
- Context Names are used whenever you need a symbolic object.

Double on GPU

```
try:
    from pygpu import gpuarray
except ImportError:
    pass


class DoubleGpu(Op, GpuKernelBase):
    __props__ = ()

    def make_node(self, x):
        ctx_name = infer_context_name(x)
        x = as_gpuarray_variable(x, ctx_name)
        return Apply(self, [x], [x.type()])

    def get_params(self, node):
        return node.outputs[0].type.context
```

Double on GPU

```
    def gpu_kernels(self, node, name):
        dt = node.inputs[0].type
        code = """
KERNEL void double(GLOBAL_MEM %(ctype) *out,
                   GLOBAL_MEM const %(ctype)s *a,
                   ga_size n) {
  for (ga_size i = LID_0; i < n; i += LDIM_0) {
    out[i] = 2 * a[i];
  }
}
""" % dict(ctype=gpuarray.dtype_to_ctype(dt))
        return [Kernel(code=code, name="double",
                       params=[gpuarray.GpuArray,
                               gpuarray.GpuArray,
                               gpuarray.SIZE],
                       flags=Kernel.get_flags(dt))]
```

Double on GPU

```
    def c_code(self, node, name, inn, outn, sub):
        return """
size_t n = 1;
Py_XDECREF(%(out)s);
%(out)s = pygpu_empty(PyGpuArray_NDIM(%(inp)s),
                      PyGpuArray_DIMS(%(inp)s),
                      GA_C_ORDER, %(ctx)s, Py_None);
if (%(out)s == NULL) %(fail)s
for (unsigned int i = 0; i < %(inp)s->ga.nd; i++)
  n *= PyGpuArray_DIM(%(inp)s, i);
if (double_scall(1, &n, 0, %(out)s, %(inp)s, n)) {
  PyErr_SetString(PyExc_RuntimeError,
                  "Error calling kernel");
  %(fail)s;
}
""" % dict(inp=inn[0], out=outn[0], fail=sub["fail"])
```

GpuKernelBase

```
class DoubleCGpu(CGpuKernelBase):
    __props__ = ()

    def __init__(self):
        CGpuKernelBase.__init__(self, ["doublecgpu.c"],
                                "double_fn")

    def make_node(self, x):
        ctx_name = infer_context_name(x)
        x = as_gpuarray_variable(x, ctx_name)
        return Apply(self, [x], [x.type()])

    def get_params(self, node):
        return node.outputs[0].type.context
```

GpuKernelBase

```
#section kernels
#kernel double : *, *, size :

KERNEL void double(GLOBAL_MEM DTYPE_o0 *out,
                   GLOBAL_MEM DTYPE_i1 *a,
                   ga_size n) {
  for (ga_size i = LID_0; i < n; i += LDIM_0) {
    out[i] = 2 * a[i];
  }
}
```

GpuKernelBase

```
#section support_code_struct
int double_fn(PyGpuArrayObject *inp, npy_int64 n,
              PyGpuArrayObject **out,
              PyGpuContextObject *ctx) {
  size_t n = 1;
  Py_XDECREF(*out);
  *out = pygpu_empty(PyGpuArray_NDIM(inp),
                     PyGpuArray_DIMS(inp),
                     GA_C_ORDER, ctx, Py_None);
  if (*out == NULL) return -1;
  for (unsigned int i = 0; i < inp->ga.nd; i++)
    n *= PyGpuArray_DIM(inp, i);
  if (double_scall(1, &n, 0, *out, inp, n)) {
    PyErr_SetString(PyExc_RuntimeError,
                    "Error calling kernel");
    return -1;
  }
```

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

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

```
from scalmulop import ScalMulV1
from doubleop import DoubleOp
from theano.gof import local_optimizer
from theano.tensor.opt import register_specialize


@register_specialize
@local_optimizer([ScalMulV1])
def local_scalmul_double(node):
    if not (isinstance(node.op, ScalMulV1) and
                node.op.scal == 2):
        return False

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

Replace an Op for GPU

Here is code to move the Double op to GPU.

```
from scalmulop import ScalMulV1
from doubleop import DoubleOp
from doublecop import DoubleCOp
from doublec import DoubleC
from doublecgpu import DoubleCGpu
from theano.gpuarray.opt import (register_opt, op_lifter,
                                  register_opt2)

@register_opt('fast_compile')
@op_lifter([DoubleOp, DoubleC, DoubleCOp])
@register_opt2([DoubleOp, DoubleC, DoubleCOp],
               'fast_compile')
def local_scalmul_double_gpu(op, context_name, inputs,
                              outputs):
    return DoubleCGpu
```

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

- Implement a ScalMulOp that multiplies its input by an arbitrary scalar value. Start with a python implementation
- Add C code to your implementation
- Create a GPU version of your op.
- Create an optimization that replace the CPU version with a GPU version when appropriate.

Clone the repo at `https://github.com/abergeron/ccw_tutorial_theano.git`.