Using ONNX and ATen to export models from PyTorch to Caffe2

When using ONNX to export a model from PyTorch into Caffe2, you sometimes end up hitting operators that are not yet part of the ONNX specification. These may be operators that haven't been standardized yet, or custom `torch.autograd.Function` types that are specific to a network.

To bridge this gap, we provide an experimental operator in ONNX that allows you to directly access PyTorch's tensor functions using the ATen library. ATen is the underlying C++ library that PyTorch uses to do tensor operations. Caffe2 has an ATen operator that can run these tensor functions in a Caffe2 network after importing them through ONNX.

This guide explains how to configure Caffe2 and modify your PyTorch program to use this functionality.

Enable ATen in Caffe2

The ATen facility in Caffe2 is part of a contrib package and needs to be enabled when you configure Caffe2 using cmake:

```
git clone https://github.com/caffe2/caffe2/
mkdir caffe2/build
cd caffe2/build
cmake -DUSE_ATEN=ON <other build options> ..
make install
```

Describe How to Export a PyTorch Autograd Function using ATen

To export a model to ONNX, PyTorch first creates a trace of all the `torch.autograd.Function`s run in the forward pass of a network. For each function in the trace, it calls that function's `symbolic` method which describes how to construct the part of the ONNX graph that will compute this function (see basic_ops.py for examples).

When equivalent ONNX operators do not exist, you can instead call any ATen function. As an example let's assume we have an autograd function which computes `x*x+y`:

```
  class MyFunction(Function):
    @staticmethod
    def forward(ctx, x, y):
      return x*x + y
```

We can add a `symbolic` method to it like so:

```
  class MyFunction(Function):
    @staticmethod
    def forward(ctx, x, y):
      return x*x + y
    @staticmethod
    def symbolic(graph, x, y):
      x2 = graph.at("mul", x, x)
      r = graph.at("add", x2, y)
      # x, y, x2, and r are 'Node' objects
      # print(r) or print(graph) will print out a textual representation for
debugging.
```

```
      # this representation will be converted to ONNX protobufs on export.
      return r
```

The function `graph.at` adds a new ATen op the computation graph. You can call any ATen function using this facility. To do so, first identify a function in ATen you want to call in Functions.h, Tensor.h, or Type.h.

As an example, we might want to call the `pow` operator:

```
static inline Tensor pow(const Tensor & self, Scalar exponent);
```

We can translate this into the equivalent `graph.at` function:

```
  def symbolic(graph, x):
    graph.at("pow", x, exponent_f = 2.0) # compute x**2
```

Tensor arguments to ATen functions become arguments to `graph.at`, while a `Scalar` like `exponent` becomes a keyword argument that specify ONNX attributes. Attributes are suffixed with their type (`_f` for floats and `_i` for integers, and `_s` for strings).

For methods, the first input is always the `this` Tensor in C++. To call methods of ATen's `Type` objects, you provide an additional string attribute that determines the type. For instance, `ones` creates a new constant tensor of all ones:

```
class Type {
  ...
  virtual Tensor ones(IntArrayRef size) const;
  ...
};
```

From PyTorch it can be created by adding the type as an additional attribute:

```
  def symbolic(graph, x):
    return graph.at("ones", type_s="float", size_i=[2,4])
```

Generally ATen operators are polymorphic across input types, and work on both the CPU and CUDA.

Putting it together

With these building blocks we can now write and export networks that include custom operators using `torch.onnx.export`:

```
class MyModule(nn.Module):
    def forward(self, x, y):
        # you can combine your ATen ops with standard onnx ones
        x = nn.ReLU()(x)
        return MyFunction.apply(x, y)

torch.onnx.export(MyModule(),
                  (Variable(torch.ones(3,4)), Variable(torch.ones(3,4))),
                  "output.onnx",
                  verbose=True)
```

This exports the following graph, which contains calls the `ATen` operator:

```
graph(%1 : Float(3, 4)
      %2 : Float(3, 4)) {
   %3 : Float(3, 4) = Relu(%1), uses = [%4.i0, %4.i1];
   %4 : UNKNOWN_TYPE = ATen[operator=mul](%3, %3), uses = [%5.i0];
   %5 : Float(3, 4) = ATen[operator=add](%4, %2), uses = [%0.i0];
   return (%5);
}
```

The graph can then be imported using ONNX and run with Caffe2:

```
import onnx
import caffe2.python.onnx.backend
import numpy as np

graph = onnx.load("output.onnx")

a = np.random.randn(3, 2).astype(np.float32)
b = np.random.randn(3, 2).astype(np.float32)

prepared_backend = caffe2.python.onnx.backend.prepare(graph)
W = {graph.graph.input[0].name: a, graph.graph.input[1].name: b}
c2_out = prepared_backend.run(W)[0]

x = np.maximum(a, 0)
r = x*x + b
np.testing.assert_array_almost_equal(r, c2_out)
```

Code

For the full source code for this tutorial, see sample.py.