An ATen operator for Caffe2

ATen is a simple tensor library thats exposes the Tensor operations in Torch and PyTorch directly in C++14. This library provides a generated wrapper around the ATen API that makes these functions available in Caffe2 as an operator. It also makes it accessible using the ToffeeIR.

Example Usage in Caffe2

First identify a function in ATen you want to call in Functions.h, Tensor.h, or Type.h.

We will call the `pow` operator:

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

Now create a Caffe2 operator to call this op. The name of the operator is always `"ATen"`, and there is always a string attribute `operator` that defines which ATen function to call:

```
import numpy as np
from caffe2.python import core, workspace


# create the Caffe2 Op:
op = core.CreateOperator(
    "ATen",
    ["MyInput"],
    ["MyOutput"],
    operator="pow", exponent=2.0)
```

Each `Tensor` input becomes an Caffe2 input Blob, and each output becomes a Caffe2 output blob. Non-tensor inputs such as `Scalar exponent` become Caffe2 `arg` attributes. In the case of `Scalar` the attributes can be either an integers or floating point numbers.

The op can now be run like any other Caffe2 operator:

```
workspace.FeedBlob("MyInput",np.random.randn(2,3).astype(np.float32))
workspace.RunOperatorOnce(op)
print(workspace.FetchBlob("MyOutput")
```

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:

```
# create a 2x4 tensor filled with floating point ones
op = core.CreateOperator(
    "ATen",
    [],
    ["MyOutput"],
    operator="ones", type="Float", size={2,4})
```

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

Example Usage via PyTorch Symbolic

The ATen operator can also be used to define `symbolic` definitions for PyTorch when an operator is being exported to ONNX. In this case, the definition of the operator looks the same but is defined using PyTorch's ONNX API:

```
class Add(torch.autograd.Function):

    @staticmethod
    def symbolic(g, a, b):
        return g.op("ATen", a, b, operator_s = "add")

    @staticmethod
    def forward(ctx, a, b):
        return a + b
```