Autograd

Autograd is a hotspot for PyTorch performance, so most of the heavy lifting is implemented in C++. This implies that we have to do some shuffling between Python and C++; and in general, we want data to be in a form that is convenient to manipulate from C++.

Our general model is that for any key data type that autograd manipulates, there are two implementations: a C++ type and a Python object type. For example, consider variables in autograd: we have both `Variable` in `variable.h` (the C++ type) and `THPVariable` in `python_variable.h` (the Python type.) (By the way, THP stands for TorcH Python, not to be confused with THPP, TorcH C++). `Variable` contains the payload of a variable, while `THPVariable` just contains a `shared_ptr` reference to `Variable`, as well as references to other Python objects which the Python runtime needs to know about. A lot of data accessor implementations in `python_variable.cpp` simply reach through to the underlying `Variable` and return the appropriate value.

The most complicated application of this principle is Function, which also supports users implementing custom behavior in Python. We have the following classes:

- `Node` in `function.h`, the C++ type.
- `THPFunction` in `python_function.h`, the Python object type. In `python_function.cpp`, you can see the boilerplate that tells the Python interpreter about this object.
- `PyNode` in `python_function.h`, a subclass of `Node` which forwards `apply` to a Python `THPFunction`. (NOT a Python object, despite its name!)

Outside of `PyNode`, the C++ objects largely avoid referencing Python objects (there are a few exceptions, like `pyobj` in `Variable`, and `PyNode`, whose whole point is to let C++ call into Python). And `pyobj` in `Node` to ensure uniqueness of the associated python wrapper (if it exists).