csrc

The csrc directory contains all of the code concerned with integration with Python. This is in contrast to lib, which contains the Torch libraries that are Python agnostic. csrc depends on lib, but not vice versa.

There are a number of utilities for easing integration with Python which are worth knowing about, which we briefly describe here. But the most important gotchas:

- DO NOT forget to take out the GIL with `pybind11::gil_scoped_acquire` before calling Python API or bringing a `THPObjectPtr` into scope.
- Make sure you include `Python.h` first in your header files, before any system headers; otherwise, you will get `error: "_XOPEN_SOURCE" redefined` error. If you pay attention to warnings, you will see where you need to do this.

Notes

Note [Storage is not nullptr]

Historically, Torch supported nullptr storage, as a minor optimization to avoid having to allocate a storage object when it would be empty. However, this is actually a confusing special case to deal with, so by-in-large, PyTorch assumes that, in fact, storage is never nullptr.

One important case where this assumption is important is when tracking the CUDA device a tensor is stored in: this information is stored solely in the storage, so if a storage is nullptr, we lose this information.

Although storage is never nullptr, the data field of c10::StorageImpl may be nullptr. This mostly occurs when we want to pre-allocate an output tensor struct, but then have it be resized and filled with data by some operator: there's no point in allocating data for it in this case!

Files

`Exceptions.h`

Frequently when working with the Python API, you may call a function which returns an error. In this case, we want to return directly to the Python interpreter, so that this exception can be propagated accordingly; however, because the Python API is C-based, what actually will happen is it will return control to whatever C++ code called it. Similarly, if we raise a C++ exception, prior to returning to the Python interpreter, we must set the Python error flags, so it turns into a C++ exception.

Moreover, when using the following macros, the generated warnings will be converted into python warnings that can be caught by the user.

Exceptions define helpers for two main cases:

- For code where you write the python binding by hand, `HANDLE_TH_ERRORS`, `END_HANDLE_TH_ERRORS` and an exception class `python_error`. You call them like this:

```
// Entry point from Python interpreter
PyObject* run(PyObject* arg) {
  HANDLE_TH_ERRORS
  ...
  if (!x) throw python_error();
  // From c10/Exception.h
  TORCH_CHECK(cond, "cond was false here");
  TORCH_WARN("Warning message");
  ...
  END_HANDLE_TH_ERRORS
}
```

The `HANDLE_TH_ERRORS` macro will catch all exceptions and convert them into an appropriate Python signal. `python_error` is a special exception which doesn't contain any info, instead it says, "An error occurred in the Python API; if you return to the interpreter, Python will raise that exception, nothing else needs to be done."

- For code that you bind using pybind, `HANDLE_TH_ERRORS` and `END_HANDLE_TH_ERRORS_PYBIND` can be used. They will work jointly with pybind error handling to raise pytorch errors and warnings natively and let pybind handle other errors. It can be used as:

```
// Function given to the pybind binding
at::Tensor foo(at::Tensor x) {
  HANDLE_TH_ERRORS
  ...
  if (!x) throw python_error();
  // pybind native error
  if (!x) throw py::value_error();
  // From c10/Exception.h
  TORCH_CHECK(cond, "cond was false here");
  TORCH_WARN("Warning message");
  ...
  END_HANDLE_TH_ERRORS_PYBIND
}
```

GIL

Whenever you make any calls to the Python API, you must have taken out the Python GIL, as none of these calls are thread safe. `pybind11::gil_scoped_acquire` is a RAII struct which handles taking and releasing the GIL. Use it like this:

```
void iWantToUsePython() {
  pybind11::gil_scoped_acquire gil;
  ...
}
```

In general, the compiler will NOT warn you if you use Python functionality without taking out the GIL, so DO NOT FORGET this call.

`utils/object_ptr.h`

`THPPointer` is a smart pointer class analogous to `std::shared_ptr`, but which is overloaded to handle reference counting scheme of various objects which are not based on `shared_ptr`. The most important overloads are:

- `PyObject` (so important we've aliased it as `THPObjectPtr`), which hooks into Python reference counting. (By the way, that means you MUST take out the GIL before bringing one of these into scope!)

- The various TH tensor and storage types (e.g., `THTensor`), which hook into TH's reference counting. (TH's reference counting IS thread safe, no locks necessary.)