PyTorch Internals Part II - The Build System

by Trevor Killeen

In the first post I explained how we generate a torch.Tensor object that you can use in your Python interpreter. Next, I will explore the build system for PyTorch. The PyTorch codebase has a variety of components:

- The core Torch libraries: TH, THC, THNN, THCUNN

- Vendor libraries: CuDNN, NCCL
- Python Extension libraries
- Additional third-party libraries: NumPy, MKL, LAPACK

How does a simple invocation of python setup.py install do the work that allows you to call import torch and use the PyTorch library in your code?

The first part of this document will explain the build process from and end-user point of view. This will explain how we take the components above to build the library. The second part of the document will be important for PyTorch developers. It will document ways to improve your iteration speed by building only a subset of the code that you are working on.

Setuptools and PyTorch's setup( ) function

Python uses Setuptools to build the library. Setuptools is an extension to the original distutils system from the core Python library. The core component of Setuptools is the setup.py file which contains all the information needed to build the project. The most important function is the setup() function which serves as the main entry point. Let's take a look at the one in PyTorch:

```
setup(name="torch", version=version,
      description="Tensors and Dynamic neural networks in Python with strong GPU acceleration",
      ext_modules=extensions,
      cmdclass={
          'build': build,
          'build_py': build_py,
          'build_ext': build_ext,
          'build_deps': build_deps,
          'build_module': build_module,
          'develop': develop,
          'install': install,
          'clean': clean,
      },
      packages=packages,
      package_data={'torch': [
          'lib/*.so*', 'lib/*.dylib*',
          'lib/torch_shm_manager',
          'lib/*.h',
          'lib/include/TH/*.h', 'lib/include/TH/generic/*.h',
          'lib/include/THC/*.h', 'lib/include/THC/generic/*.h']},
      install_requires=['pyyaml'],
      )
```

The function is composed entirely of keyword arguments, which serve two purposes:

- Metadata (e.g. name, description, version)
- The contents of the package

We are concerned with #2. Let's break down the individual components:

- **ext_modules**: Python modules are either "pure" modules, containing only Python code, or "extension" modules written in the low-level language of the Python implementation. Here we are listing the extension modules in the build, including the main torch._C library that contains our Python Tensor
- **cmdclass**: When using the setup.py script from the command line, the user must specify one or more "commands", code snippets that perform a specific action. For example, the "install" command builds and installs the package. This mapping routes specific commands to functions in setup.py that implement them
- **packages**: The list of packages in the project. These are "pure" - i.e. they only contain Python code. These are defined elsewhere in setup.py
- **package_data**: Additional files that need to be installed into a package: in this case the header files and shared libraries that the build will generate must be included in our installation
- **install_requires**: In order to build PyTorch, we need pyyaml. Setuptools will handle making sure that pyyaml will be available, downloading and installing it if necessary

We will consider these components in more detail, but for now it is instructive to look at the end product of an installation – i.e. what Setuptools does after building the code.

site_packages

Third party packages are by default installed into the lib/<version>/site_packages directory associated with your Python binary. For example, because I am using an Miniconda environment, my Python binary is found at:

```
(p3) killeent@devgpu047:pytorch (master)$ which python
~/local/miniconda2/envs/p3/bin/python
```

And thus packages are installed into:

```
/home/killeent/local/miniconda2/envs/p3/lib/python3.6/site-packages
```

I installed PyTorch, and let's take a look into torch folder in site-packages:

```
(p3) killeent@devgpu047:site-packages$ cd torch
(p3) killeent@devgpu047:torch$ ls
autograd  backends  _C.cpython-36m-x86_64-linux-gnu.so  cuda  distributed  _dl.cpython-36m-x86_64-linux-gnu.so  functional.py  __in
```

Note that everything we would expect to be here is here:

- All the "pure" packages are here [todo print packages from setup.py to explain]
- The extension libraries are here - the ._C* and ._dl* shared libraries

- The package_data is here: the contents of lib/ match exactly what we described in the setup function:

```
(p3) killeent@devgpu047:torch$ ls lib/
include     libnccl.so.1  libTHC.so.1   libTHCUNN.so.1  libTHNN.so.1  libTH.so.1   THCUNN.h  torch_shm_manager libnccl.so  libshm.s
```

The Python interpreter looks into site_packages during an import. If we call import torch in our Python code it will find the module here and initialize and import it. You can read more about the import system here.

Building Individual Parts

Next, we will look at the various individual components of the build from start to finish. This will illustrate how we combine all the code we mentioned in the introduction.

Backend Torch and Vendor Libraries

Let's take a look at the install cmd override in PyTorch's setup.py:

```
class install(setuptools.command.install.install):

    def run(self):
        if not self.skip_build:
            self.run_command('build_deps')
        setuptools.command.install.install.run(self)
```

We note the first thing it does is run a command called "build_deps" - let's take a look at it's run() method:

```
def run(self):
        from tools.nnwrap import generate_wrappers as generate_nn_wrappers
        build_all_cmd = ['bash', 'torch/lib/build_all.sh']
        if WITH_CUDA:
            build_all_cmd += ['--with-cuda']
        if WITH_NCCL and not SYSTEM_NCCL:
            build_all_cmd += ['--with-nccl']
        if WITH_DISTRIBUTED:
            build_all_cmd += ['--with-distributed']
        if subprocess.call(build_all_cmd) != 0:
            sys.exit(1)
        generate_nn_wrappers()
```

Here we note that that we have a shell script build_all.sh in the torch/lib/ directory. This script is configurable by whether we are on a system with CUDA enabled, the NCCL library enabled, and PyTorch's distributed library enabled.

Let's take a look in torch/lib:

```
(p3) killeent@devgpu047:lib (master)$ ls
build_all.sh  libshm  nccl  README.md  TH  THC  THCS  THCUNN  THD  THNN  THPP  THS
```

Here we see the directories for all the backend libraries. TH, THC, THNN, THCUNN, and nccl are git subtrees that are in sync with the libraries in e.g. github.com/torch. THS, THCS, THD, THPP and libshm are libraries specific to PyTorch. All of the libraries contain CMakeLists.txt - indicating they are built with CMake.

The build_all.sh is essentially a script that runs the CMake configure step on all of these libraries, and then make install. Let's run ./build_all.sh and see what we are left with:

```
(p3) killeent@devgpu047:lib (master)$ ./build_all.sh --with-cuda --with-nccl --with-distributed
[various CMake output logs]
(p3) killeent@devgpu047:lib (master)$ ls
build  build_all.sh  include  libnccl.so  libnccl.so.1  libshm  libshm.so  libTHC.so.1  libTHCS.so.1  libTHCUNN.so.1  libTHD.so.1
```

Now there are a number of extra things in the directory:

- Shared library files for each library
- Headers for THNN and THCUNN
- build and tmp_install directories
- The torch_shm_manager executable

Let's explore further. In the shell script, we create the build directory and a subdir for each library to build:

```
# We create a build directory for the library, which will
# contain the cmake output. $1 is the library to be built
  mkdir -p build/$1
  cd build/$1
```

Thus e.g. build/TH contains the CMake configuration output including the Makefile for building TH, and also the result of running make install in this directory.

Let's also look at tmp_install:

```
(p3) killeent@devgpu047:lib (master)$ ls tmp_install/
bin  include  lib  share
```

tmp_install looks like a standard install directory containing binaries, header files and library files. For example, tmp_install/include/TH contains all the TH headers, and tmp_install/lib/ contains the libTH.so.1 file.

So why have this directory? It is used to compile the libraries that depend on each other. For example, the THC library depends on the TH library and its headers. This is referenced in the build shell script as arguments to the cmake command:

```
# install_dir is tmp_install
cmake ...
```

```
        -DTH_INCLUDE_PATH="$INSTALL_DIR/include" \
        -DTH_LIB_PATH="$INSTALL_DIR/lib" \
```

And indeed if we look at the THC library we built:

```
(p3) killeent@devgpu047:lib (master)$ ldd libTHC.so.1
        ...
        libTH.so.1 => /home/killeent/github/pytorch/torch/lib/tmp_install/lib/./libTH.so.1 (0x00007f84478b7000)
```

The way the build_all.sh specifies the include and library paths is a little messy but this is representative of the overall idea. Finally, at the end of the script:

```
# If all the builds succeed we copy the libraries, headers,
# binaries to torch/lib
cp $INSTALL_DIR/lib/* .
cp THNN/generic/THNN.h .
cp THCUNN/generic/THCUNN.h .
cp -r $INSTALL_DIR/include .
cp $INSTALL_DIR/bin/* .
```

As we can see, at the end, we copy everything to the top-level torch/lib directory - explaining the contents we saw above. We'll see why we do this next:

NN Wrappers

Briefly, let's touch on the last part of the build_deps command: generate_nn_wrappers(). We bind into the backend libraries using PyTorch's custom cwrap tooling, which we touched upon in a previous post. For binding TH and THC we manually write the YAML declarations for each function. However, due to the relative simplicity of the THNN and THCUNN libraries, we auto-generate both the cwrap declarations and the resulting C++ code.

The reason we copy the THNN.h and THCUNN.h header files into torch/lib is that this is where the generate_nn_wrappers() code expects these files to be located. generate_nn_wrappers() does a few things:

1. Parses the header files, generating cwrap YAML declarations and writing them to output .cwrap files
2. Calls cwrap with the appropriate plugins on these .cwrap files to generate source code for each
3. Parses the headers *a second time* to generate THNN_generic.h - a library that takes THPP Tensors, PyTorch's "generic" C++ Tensor Library, and calls into the appropriate THNN/THCUNN library function based on the dynamic type of the Tensor

If we take a look into torch/csrc/nn after running generate_nn_wrappers() we can see the output:

```
(p3) killeent@devgpu047:nn (master)$ ls
THCUNN.cpp  THCUNN.cwrap  THNN.cpp  THNN.cwrap  THNN_generic.cpp  THNN_generic.cwrap  THNN_generic.h  THNN_generic.inc.h
```

For example, the code generates cwrap like:

```
[[
  name: FloatBatchNormalization_updateOutput
  return: void
  cname: THNN_FloatBatchNormalization_updateOutput
  arguments:
    - void* state
    - THFloatTensor* input
    - THFloatTensor* output
    - type: THFloatTensor*
      name: weight
      nullable: True
    - type: THFloatTensor*
      name: bias
      nullable: True
    - THFloatTensor* running_mean
    - THFloatTensor* running_var
    - THFloatTensor* save_mean
    - THFloatTensor* save_std
    - bool train
    - double momentum
    - double eps
]]
```

with corresponding .cpp:

```
extern "C" void THNN_FloatBatchNormalization_updateOutput(void*, THFloatTensor*, THFloatTensor*, THFloatTensor*, THFloatTensor*, TH

PyObject * FloatBatchNormalization_updateOutput(PyObject *_unused, PyObject *args) {
        // argument checking, unpacking
         PyThreadState *_save = NULL;
      try {
        Py_UNBLOCK_THREADS;
        THNN_FloatBatchNormalization_updateOutput(arg_state, arg_input, arg_output, arg_weight, arg_bias, arg_running_mean, arg_run
        Py_BLOCK_THREADS;
        Py_RETURN_NONE;
      } catch (...) {
        if (_save) {
          Py_BLOCK_THREADS;
        }
        throw;
      }
```

```
    ...
}
```

In the THPP generated code, the function looks like this:

```
void BatchNormalization_updateOutput(thpp::Tensor* input, thpp::Tensor* output, thpp::Tensor* weight, thpp::Tensor* bias, thpp::Ten
        // Call appropriate THNN function based on tensor type, whether its on CUDA, etc.
}
```

We will look a little more at how these source files are used later.

“Building” the Pure Python Modules

Now that we have built the backend libraries (the “dependencies”) we can move forward with building the actual PyTorch code. The next Setuptools command that runs is build_py, which is used to build all the “Pure” python modules in our library. These are the “packages” passed to setup.py.

The packages are found using the Setuptools’ utility function find_packages():

```
packages = find_packages(exclude=('tools.*',))
['torch', 'torch._thnn', 'torch.autograd', 'torch.backends', 'torch.cuda', 'torch.distributed', 'torch.legacy', 'torch.multiprocess
```

As we can see, find_package has recursively traversed the torch directory, finding all the directory paths that have an __init__.py file.

When building with Setuptools, the tool creates a build directory in the distribution root, i.e. the same location as the setup.py file. Because PyTorch is composed of both “Pure” python modules and Extension Modules, we need to preserve information about the Operating System and Python version used when performing the build. So if we look in my build directory, we see:

```
(p3) killeent@devgpu047:pytorch (master)$ ls build
lib.linux-x86_64-3.6  temp.linux-x86_64-3.6
```

This indicates that I’ve built the project on linux-x86-64 using Python 3.6. The lib directory contains the library files, while the temp directory contains files generated during the build that aren’t needed in the final installation.

Because “Pure” python modules are just Python code, and don’t need to be “compiled”, the build_py process simply copies files from their locations as found by find_packages to the equivalent location in build/. So our build output is littered with lines like:

```
copying torch/autograd/_functions/blas.py -> build/lib.linux-x86_64-3.6/torch/autograd/_functions
```

We also noted earlier that we could pass files and directories to the package_data keyword argument to the main setup() function, and that Setuptools would handle copying those files to the installation location. During build_py, these files are copied to the build/ directory, so we also see lines like:

```
copying torch/lib/libTH.so.1 -> build/lib.linux-x86_64-3.6/torch/lib
...
copying torch/lib/include/THC/generic/THCTensor.h -> build/lib.linux-x86_64-3.6/torch/lib/include/THC/generic
```

Building the Extension Modules

Finally, we need to build the Extension Modules, i.e. the PyTorch modules written in C++ using the CPython backend. This also constitutes the majority of the code logic in setup.py. Our overridden build_ext Command has some special logic before the extensions themselves are actually built:

```
from tools.cwrap import cwrap
from tools.cwrap.plugins.THPPlugin import THPPlugin
from tools.cwrap.plugins.ArgcountSortPlugin import ArgcountSortPlugin
from tools.cwrap.plugins.AutoGPU import AutoGPU
from tools.cwrap.plugins.BoolOption import BoolOption
from tools.cwrap.plugins.KwargsPlugin import KwargsPlugin
from tools.cwrap.plugins.NullableArguments import NullableArguments
from tools.cwrap.plugins.CuDNNPlugin import CuDNNPlugin
from tools.cwrap.plugins.WrapDim import WrapDim
from tools.cwrap.plugins.AssertNDim import AssertNDim
from tools.cwrap.plugins.Broadcast import Broadcast
from tools.cwrap.plugins.ProcessorSpecificPlugin import ProcessorSpecificPlugin
        thp_plugin = THPPlugin()
        cwrap('torch/csrc/generic/TensorMethods.cwrap', plugins=[
            ProcessorSpecificPlugin(), BoolOption(), thp_plugin,
            AutoGPU(condition='IS_CUDA'), ArgcountSortPlugin(), KwargsPlugin(),
            AssertNDim(), WrapDim(), Broadcast()
        ])
        cwrap('torch/csrc/cudnn/cuDNN.cwrap', plugins=[
            CuDNNPlugin(), NullableArguments()
        ])
```

Recall above that I documented that we auto-generated C++ code for calling into the THNN etc. libraries. Here is where we bind TH, THC and CuDNN. We take the YAML declarations in TensorMethods.cwrap, and use them to generate output C++ source files that contain implementations that work within PyTorch’s C++ Ecosystem. For example, a simple declaration like zero_:

```
[[
  name: zero_
  cname: zero
  return: self
  arguments:
```

```
    - THTensor* self
]]

Generates code like:
 PyObject * THPTensor_(zero_)(PyObject *self, PyObject *args, PyObject *kwargs) {
        ...
        THTensor_(zero)(LIBRARY_STATE arg_self);
        ...
}
```

In the previous post we documented how these functions are tied to specific Tensor types, so I won't expand on that there. For the build process its enough to know that these C++ files are generated prior to the extension being built, because these source files are used during Extension compilation.

Specifying the Extensions

Unlike pure modules, it's not enough just to list modules or packages and expect the Setuptools to go out and find the right files; you have to specify the extension name, source file(s), and any compile/link requirements (include directories, libraries to link with, etc.).

The bulk (200~ LOC at the time of this writing) of the setup.py goes into specifying how to build these Extensions. Here, some of the choices we make in build_all.sh begin to make sense. For example, we saw that our build script specified a tmp_install directory where we installed our backend libraries. In our setup.py code, we reference this directory when adding to the list of directories containing header files to include:

```
# tmp_install_path is torch/lib/tmp_install
include_dirs += [
    cwd,
    os.path.join(cwd, "torch", "csrc"),
    tmp_install_path + "/include",
    tmp_install_path + "/include/TH",
    tmp_install_path + "/include/THPP",
    tmp_install_path + "/include/THNN",
```

Similarly, we copied the shared object libraries to torch/csrc at the end of the build_all.sh script. We reference these locations directly in our setup.py code when identifying libraries that we may link against:

```
# lib_path is torch/lib
TH_LIB = os.path.join(lib_path, 'libTH.so.1')
THS_LIB = os.path.join(lib_path, 'libTHS.so.1')
THC_LIB = os.path.join(lib_path, 'libTHC.so.1')
THCS_LIB = os.path.join(lib_path, 'libTHCS.so.1')
THNN_LIB = os.path.join(lib_path, 'libTHNN.so.1')
# ...
```

Let's consider how we build the main torch._C Extension Module:

```
C = Extension("torch._C",
              libraries=main_libraries,
              sources=main_sources,
              language='c++',
              extra_compile_args=main_compile_args + extra_compile_args,
              include_dirs=include_dirs,
              library_dirs=library_dirs,
              extra_link_args=extra_link_args + main_link_args + [make_relative_rpath('lib')],
              )
```

- The main libraries are all the libraries we link against. This includes things like shm, PyTorch's shared memory management library, and also system libraries like cudart and cudnn. Note that the TH libraries *are not* listed here
- The main sources are the C++ files that make up the C++ backend for PyTorch
- The compile args are various flags that configure compilation. For example, we might want to add debug flags when compiling in debug mode
- The include dirs are the paths to all the directories containing header files. This is also another example where the build_all.sh script is important - for example, we look for the TH header files in torch/lib/tmp_install/include/TH - which is the install location we specified with our CMake configuration
- The library dirs are directories to search for shared libraries at link time. For example, we include torch/lib - the location we copied our .so files to at the end of build_all.sh, but also the paths to the CUDA and CuDNN directories
- The link arguments are used when linking object files together to create the extension. In PyTorch, this includes more *normal* options like decided to link libstdc++ statically. However, there is one key component: **this is where we link the backend TH libraries**. Note that we have lines like:

```
# The explicit paths to .so files we described above
main_link_args = [TH_LIB, THS_LIB, THPP_LIB, THNN_LIB]
```

You might be wondering why we do this as opposed to adding these libraries to the list we pass to the libraries keyword argument. After all, that is a list of libraries to link against. The issue is that Lua Torch installs often set the LD_LIBRARY_PATH variable, and thus we could mistakenly link against a TH library built for Lua Torch, instead of the library we have built locally. This would be problematic because the code could be out of date, and also there are various configuration options for Lua Torch's TH that would not play nicely with PyTorch.

As such, we manually specify the paths to the shared libraries we generated directly to the linker.

There are other extensions needed to power PyTorch and they are built in a similar way. The Setuptools library invokes the C++ compiler and linker to build all of these extensions. If the builds succeed, we have successfully *built* the PyTorch library and we can move on to installation.

Installation

After building has finished, installation is quite simple. We simply have to copy everything from our build/lib.linux-x86_64-3.6 directory to the appropriate installation directory. Recall that we noted above that this directory is the site_packages directory associated with our Python binary. As a result, we see lines like:

```
running install_lib
creating /home/killeent/local/miniconda2/envs/p3/lib/python3.6/site-packages/torch
copying build/lib.linux-x86_64-3.6/torch/_C.cpython-36m-x86_64-linux-gnu.so -> /home/killeent/local/miniconda2/envs/p3/lib/python3.
copying build/lib.linux-x86_64-3.6/torch/_dl.cpython-36m-x86_64-linux-gnu.so -> /home/killeent/local/miniconda2/envs/p3/lib/python3
creating /home/killeent/local/miniconda2/envs/p3/lib/python3.6/site-packages/torch/_thnn
copying build/lib.linux-x86_64-3.6/torch/_thnn/_THNN.cpython-36m-x86_64-linux-gnu.so -> /home/killeent/local/miniconda2/envs/p3/lib
copying build/lib.linux-x86_64-3.6/torch/_thnn/_THCUNN.cpython-36m-x86_64-linux-gnu.so -> /home/killeent/local/miniconda2/envs/p3/l
```

Finally lets power up the Python interpreter. When the Python interpreter executes an import statement, it searches for Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built.

```
# note we are now in my home directory
(p3) killeent@devgpu047:~$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, Mar 22 2017, 19:54:23)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/home/killeent/local/miniconda2/envs/p3/lib/python36.zip', '/home/killeent/local/miniconda2/envs/p3/lib/python3.6', '/home/ki
```

As we can see, the site-packages directory we copied our PyTorch installation to is part of search path. Now let's load the torch module and see its location:

```
>>> import torch
>>> import inspect
>>> inspect.getfile(torch)
'/home/killeent/local/miniconda2/envs/p3/lib/python3.6/site-packages/torch/__init__.py'
```

As we can see, we have loaded the module from site_packages as expected - and our build and installation is successful!

Note: Python prepends the empty string to sys.path to represent the current working directory - making it the first place we search for a module. So if we run Python from the pytorch directory, we would accidentally load the local version of PyTorch rather than our installed version. This is something to watch out for.

Addendum - Developer Efficiency, 3rd Party Libraries, Things I Didn't Cover

The entire installation loop for PyTorch can be quite time-consuming. On my devserver, it takes around 5 minutes for an installation from source. Often times, when developing PyTorch, we only want to work on a subset of the entire project, and re-build only that subset in order to test changes. Fortunately, our build system enables this.

Setuptools Develop Mode

The main tool that supports this is Setuptools develop command. The documentation states that:

> *This command allows you to deploy your project's source for use in one or more "staging areas" where it will be available for importing. This deployment is done in such a way that changes to the project source are immediately available in the staging area(s), without needing to run a build or install step after each change.*

But how does it work? Suppose we run python setup.py build develop in the PyTorch directory. The build command is run, building our dependencies (TH, THPP, etc.) and the extension libraries. However, if we look inside site-packages:

```
(p3) killeent@devgpu047:site-packages$ ls -la torch*
-rw-r--r--. 1 killeent users 31 Jun 27 08:02 torch.egg-link
```

Looking at the contents of the torch.egg-link file, it simply references the PyTorch directory:

```
(p3) killeent@devgpu047:site-packages$ cat torch.egg-link
/home/killeent/github/pytorch
```

If we navigate back to the PyTorch directory, we see there is a new directory torch.egg-info:

```
(p3) killeent@devgpu047:pytorch (master)$ ls -la torch.egg-info/
total 28
drwxr-xr-x.  2 killeent users  4096 Jun 27 08:09 .
drwxr-xr-x. 10 killeent users  4096 Jun 27 08:01 ..
-rw-r--r--.  1 killeent users     1 Jun 27 08:01 dependency_links.txt
-rw-r--r--.  1 killeent users   255 Jun 27 08:01 PKG-INFO
-rw-r--r--.  1 killeent users     7 Jun 27 08:01 requires.txt
-rw-r--r--.  1 killeent users 16080 Jun 27 08:01 SOURCES.txt
-rw-r--r--.  1 killeent users    12 Jun 27 08:01 top_level.txt
```

This file contains metadata about the PyTorch project. For example, requirements.txt lists all of the dependencies for setting up PyTorch:

```
(p3) killeent@devgpu047:pytorch (master)$ cat torch.egg-info/requires.txt
pyyaml
```

Without going into too much detail, develop allows us to essentially treat the PyTorch repo itself as if it were in site-packages, so we can import the module and it just works:

```
(p3) killeent@devgpu047:~$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, Mar 22 2017, 19:54:23)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__file__
'/home/killeent/github/pytorch/torch/__init__.py'
```

As a result, the following consequences hold:

- If we change a Python source file, the changes are automatically picked up, and we don't have to run any commands to let the Python interpreter *see* this change
- If we change a C++ Source File in one of the extension libraries, we can re-run the develop command, it will re-build the extension

Thus we can develop the PyTorch codebases seamlessly, and test our changes in an easy way.

Working on the Dependency Libraries

If we are working on the dependencies (e.g. TH, THPP, etc.) we can re-build our changes more quickly by simply running the build_deps command directly. This will automatically call into build_all.sh to re-build our libraries, and copy the generated libraries appropriately. If we are using Setuptools develop mode, we will be using the local extension library built in the PyTorch directory. Because we have specified the paths to the shared libraries when compiling our Extension Libraries, the changes will be picked up:

```
# we are using the local extension
(p3) killeent@devgpu047:~$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, Mar 22 2017, 19:54:23)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch._C.__file__
'/home/killeent/github/pytorch/torch/_C.cpython-36m-x86_64-linux-gnu.so'

# it references the local shared object library we just re-built
(p3) killeent@devgpu047:~$ ldd /home/killeent/github/pytorch/torch/_C.cpython-36m-x86_64-linux-gnu.so
# ...
libTH.so.1 => /home/killeent/github/pytorch/torch/lib/libTH.so.1 (0x00007f543d0e2000)
# ...
```

As such, we can test any changes here without having to do a full rebuild.

3rd Party Libraries

PyTorch has dependencies on some 3rd party libraries. The usual mechanism for using these libraries is to install them via Anaconda, and then link against them. For example, we can use the mkl library with PyTorch by doing:

```
# installed to miniconda2/envs/p3/lib/libmkl_intel_lp64.so
conda install mkl
```

And then as long as we have the path to this lib directory on our $CMAKE_PREFIX_PATH, it will successfully find this library when compiling:

```
# in the site-packages dir
(p3) killeent@devgpu047:torch$ ldd _C.cpython-36m-x86_64-linux-gnu.so
# ...
libmkl_intel_lp64.so => /home/killeent/local/miniconda2/envs/p3/lib/libmkl_intel_lp64.so (0x00007f3450bba000)
# ...
```

Not Covered, But Also Relevant

- How ccache is used to speed up build times
- How PyTorch's top-level __init__.py file handles the initial module import and pulling together all the various modules and extension libraries
- The CMake build system, how the backend libraries are configured and built with CMake