A Tour of PyTorch Internals (Part I)

by Trevor Killeen

The fundamental unit in PyTorch is the Tensor. This post will serve as an overview for how we implement Tensors in PyTorch, such that the user can interact with it from the Python shell. In particular, we want to answer four main questions:

- How does PyTorch extend the Python interpreter to define a Tensor type that can be manipulated from Python code?
- How does PyTorch wrap the C libraries that actually define the Tensor's properties and methods?
- How does PyTorch cwrap work to generate code for Tensor methods?
- How does PyTorch's build system take all of these components to compile and generate a workable application?

Extending the Python Interpreter

PyTorch defines a new package torch. In this post we will consider the ._C module. This module is known as an "extension module" - a Python module written in C. Such modules allow us to define new built-in object types (e.g. the Tensor) and to call C/C++ functions.

The ._C module is defined in torch/csrc/Module.cpp. The init_C() / PyInit__C() function creates the module and adds the method definitions as appropriate. This module is passed around to a number of different __init() functions that add further objects to the module, register new types, etc.

One collection of these __init() calls is the following:
```
ASSERT_TRUE(THPDoubleTensor_init(module));
ASSERT_TRUE(THPFloatTensor_init(module));
ASSERT_TRUE(THPHalfTensor_init(module));
ASSERT_TRUE(THPLongTensor_init(module));
ASSERT_TRUE(THPIntTensor_init(module));
ASSERT_TRUE(THPShortTensor_init(module));
ASSERT_TRUE(THPCharTensor_init(module));
ASSERT_TRUE(THPByteTensor_init(module));
```

These __init() functions add the Tensor object for each type to the ._C module so that they can be used in the module. Let's learn how these methods work.

The THPTensor Type

Much like the underlying TH and THC libraries, PyTorch defines a "generic" Tensor which is then specialized to a number of different types. Before considering how this specialization works, let's first consider how defining a new type in Python works, and how we create the generic THPTensor type.

The Python runtime sees all Python objects as variables of type PyObject *, which serves as a "base type" for all Python objects. Every Python type contains the refcount for the object, and a pointer to the object's *type object*. The type object determines the properties of the type. For example, it might contain a list of methods associated with the type, and which C functions get called to implement those methods. The object also contains any fields necessary to represent its state.

The formula for defining a new type is as follows:

- Create a struct that defines what the new object will contain
- Define the type object for the type

The struct itself could be very simple. Inn Python, all floating point types are actually objects on the heap. The Python float struct is defined as:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

The PyObject_HEAD is a macro that brings in the code that implements an object's reference counting, and a pointer to the corresponding type object. So in this case, to implement a float, the only other "state" needed is the floating point value itself.

Now, let's see the struct for our THPTensor type:

```
struct THPTensor {
    PyObject_HEAD
    THTensor *cdata;
};
```

Pretty simple, right? We are just wrapping the underlying TH tensor by storing a pointer to it.

The key part is defining the "type object" for a new type. An example definition of a type object for our Python float takes the form:

```
static PyTypeObject py_FloatType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "py.FloatObject",          /* tp_name */
    sizeof(PyFloatObject),     /* tp_basicsize */
    0,                         /* tp_itemsize */
    0,                         /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_as_async */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash  */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,        /* tp_flags */
    "A floating point number", /* tp_doc */
};
```

The easiest way to think of a *type object* is as a set of fields which define the properties of the object. For example, the tp_basicsize field is set to sizeof(PyFloatObject). This is so that Python knows how much memory to allocate when calling PyObject_New() for a PyFloatObject. The full list of fields you can set is defined in object.h in the CPython backend: https://github.com/python/cpython/blob/master/Include/object.h.

The type object for our THPTensor is THPTensorType, defined in csrc/generic/Tensor.cpp. This object defines the name, size, mapping methods, etc. for a THPTensor.

As an example, let's take a look at the tp_new function we set in the PyTypeObject:

```
PyTypeObject THPTensorType = {
  PyVarObject_HEAD_INIT(NULL, 0)
  ...
  THPTensor_(pynew), /* tp_new */
};
```

The tp_new function enables object creation. It is responsible for creating (as opposed to initializing) objects of that type and is equivalent to the __new__() method at the Python level. The C implementation is a static method that is passed the type being instantiated and any arguments, and returns a newly created object.

```
static PyObject * THPTensor_(pynew)(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
  HANDLE_TH_ERRORS
  Py_ssize_t num_args = args ? PyTuple_Size(args) : 0;

  THPTensorPtr self = (THPTensor *)type->tp_alloc(type, 0);
// more code below
```

The first thing our new function does is allocate the THPTensor. It then runs through a series of initializations based off of the args passed to the function. For example, when creating a THPTensor *x* from another THPTensor *y*, we set the newly created THPTensor's cdata field to be the result of calling THTensor_(newWithTensor) with the *y*'s underlying TH Tensor as an argument. Similar constructors exist for sizes, storages, NumPy arrays, and sequences.

** Note that we solely use tp_new, and not a combination of tp_new and tp_init (which corresponds to the __init__() function).

The other important thing defined in Tensor.cpp is how indexing works. PyTorch Tensors support Python's **Mapping Protocol**. This allows us to do things like:

```
x = torch.Tensor(10).fill_(1)
y = x[3] // y == 1
x[4] = 2
// etc.
```

** Note that this indexing extends to Tensor with more than one dimension

We are able to use the []-style notation by defining the three mapping methods described here.

The most important methods are THPTensor_(getValue) and THPTensor_(setValue) which describe how to index a Tensor, for returning a new Tensor/Scalar, or updating the values of an existing Tensor in place. Read through these implementations to better understand how PyTorch supports basic tensor indexing.

Generic Builds (Part One)

We could spend a ton of time exploring various aspects of the THPTensor and how it relates to defining a new Python object. But we still need to see how the THPTensor_(init)() function is translated to the THPIntTensor_init() we used in our module initialization. How do we take our Tensor.cpp file that defines a "generic" Tensor and use it to generate Python objects for all the permutations of types? To put it another way, Tensor.cpp is littered with lines of code like:

```
return THPTensor_(New)(THTensor_(new)(LIBRARY_STATE_NOARGS));
```

This illustrates both cases we need to make type-specific:

- Our output code will call THP<Type>Tensor_New(...) in place of THPTensor_(New)
- Our output code will call TH<Type>Tensor_new(...) in place of THTensor_(new)

In other words, for all supported Tensor types, we need to “generate” source code that has done the above substitutions. This is part of the “build” process for PyTorch. PyTorch relies on Setuptools (https://setuptools.readthedocs.io/en/latest/) for building the package, and we define a setup.py file in the top-level directory to customize the build process.

One component building an Extension module using Setuptools is to list the source files involved in the compilation. However, our csrc/generic/Tensor.cpp file is not listed! So how does the code in this file end up being a part of the end product?

Recall that we are calling the THPTensor* functions (such as init) from the directory above generic. If we take a look in this directory, there is another file Tensor.cpp defined. The last line of this file is important:

```
//generic_include TH torch/csrc/generic/Tensor.cpp
```

Note that this Tensor.cpp file is included in setup.py, but it is wrapped in a call to a Python helper function called split_types. This function takes as input a file, and looks for the “//generic_include” string in the file contents. If it is found, it generates a new output file for each Tensor type, with the following changes:

- The output file is renamed to Tensor<Type>.cpp
- The output file is slightly modified as follows:

```
# Before:
//generic_include TH torch/csrc/generic/Tensor.cpp

# After:
#define TH_GENERIC_FILE "torch/src/generic/Tensor.cpp"
#include "TH/THGenerate<Type>Type.h"
```

Including the header file on the second line has the side effect of including the source code in Tensor.cpp with some additional context defined. Let’s take a look at one of the headers:

```
#ifndef TH_GENERIC_FILE
#error "You must define TH_GENERIC_FILE before including THGenerateFloatType.h"
#endif

#define real float
#define accreal double
#define TH_CONVERT_REAL_TO_ACCREAL(_val) (accreal)(_val)
#define TH_CONVERT_ACCREAL_TO_REAL(_val) (real)(_val)
#define Real Float
#define THInf FLT_MAX
#define TH_REAL_IS_FLOAT
#line 1 TH_GENERIC_FILE
#include TH_GENERIC_FILE
#undef accreal
#undef real
#undef Real
#undef THInf
#undef TH_REAL_IS_FLOAT
#undef TH_CONVERT_REAL_TO_ACCREAL
#undef TH_CONVERT_ACCREAL_TO_REAL
```

```
#ifndef THGenerateManyTypes
#undef TH_GENERIC_FILE
#endif
```

What this is doing is bringing in the code from the generic Tensor.cpp file and surrounding it with the following macro definitions. For example, we define real as a float, so any code in the generic Tensor implementation that refers to something as a real will have that real replaced with a float. In the corresponding file THGenerateIntType.h, the same macro would replace real with int.

These output files are returned from split_types and added to the list of source files, so we can see how the .cpp code for different types is created.

There are a few things to note here: First, the split_types function is not strictly necessary. We could wrap the code in Tensor.cpp in a single file, repeating it for each type. The reason we split the code into separate files is to speed up compilation. Second, what we mean when we talk about the type replacement (e.g. replace real with a float) is that the C preprocessor will perform these substitutions during compilation. Merely surrounding the source code with these macros has no side effects until preprocessing.

Generic Builds (Part Two)

Now that we have source files for all the Tensor types, we need to consider how the corresponding header declarations are created, and also how the conversions from THTensor_(method) and THPTensor_(method) to TH<Type>Tensor_method and THP<Type>Tensor_method work. For example, csrc/generic/Tensor.h has declarations like:

```
THP_API PyObject * THPTensor_(New)(THTensor *ptr);
```

We use the same strategy for generating code in the source files for the headers. In csrc/Tensor.h, we do the following:

```
#include "generic/Tensor.h"
#include <TH/THGenerateAllTypes.h>

#include "generic/Tensor.h"
#include <TH/THGenerateHalfType.h>
```

This has the same effect, where we draw in the code from the generic header, wrapped with the same macro definitions, for each type. The only difference is that the resulting code is contained all within the same header file, as opposed to being split into multiple source files.

Lastly, we need to consider how we “convert” or “substitute” the function types. If we look in the same header file, we see a bunch of #define statements, including:

```
#define THPTensor_(NAME)            TH_CONCAT_4(THP,Real,Tensor_,NAME)
```

This macro says that any string in the source code matching the format THPTensor_(NAME) should be replaced with THPRealTensor_NAME, where Real is derived from whatever the symbol Real is #define‘d to be at the time. Because our header code and source code is surrounded by macro definitions for all the types as seen above, after the preprocessor has run, the resulting code is what we would expect. The code in the TH library defines the same macro for THTensor_(NAME), supporting the translation of those functions as well. In this way, we end up with header and source files with specialized code.

Module Objects and Type Methods

Now we have seen how we have wrapped TH’s Tensor definition in THP, and generated THP methods such as THPFloatTensor_init(...). Now we can explore what the above code actually does in terms of the module we are creating. The key line in THPTensor_(init) is:

```
# THPTensorBaseStr, THPTensorType are also macros that are specific
# to each type
PyModule_AddObject(module, THPTensorBaseStr, (PyObject *)&THPTensorType);
```

This function registers our Tensor objects to the extension module, so we can use THPFloatTensor, THPIntTensor, etc. in our Python code.

Just being able to create Tensors isn’t very useful - we need to be able to call all the methods that TH defines. A simple example shows calling the in-place zero_ method on a Tensor.

```
x = torch.FloatTensor(10)
x.zero_()
```

Let’s start by seeing how we add methods to newly defined types. One of the fields in the “type object” is tp_methods. This field holds an array of method definitions (PyMethodDefs) and is used to associate methods (and their underlying C/C++ implementations) with a type. Suppose we wanted to define a new method on our PyFloatObject that replaces the value. We could implement this as follows:

```
static PyObject * replace(PyFloatObject *self, PyObject *args) {
        double val;
        if (!PyArg_ParseTuple(args, "d", &val))
                return NULL;
        self->ob_fval = val;
        Py_RETURN_NONE
}
```

This is equivalent to the Python method:

```
def replace(self, val):
        self.ob_fval = val
```

It is instructive to read more about how defining methods works in CPython. In general, methods take as the first parameter the instance of the object, and optionally parameters for the positional arguments and keyword arguments. This static function is registered as a method on our float:

```
static PyMethodDef float_methods[] = {
        {"replace", (PyCFunction)replace, METH_VARARGS,
        "replace the value in the float"
        },
        {NULL} /* Sentinel */
}
```

This registers a method called replace, which is implemented by the C function of the same name. The METH_VARARGS flag indicates that the method takes a tuple of arguments representing all the arguments to the function. This array is set to the tp_methods field of the type object, and then we can use the replace method on objects of that type.

We would like to be able to call all of the methods for TH tensors on our THP tensor equivalents. However, writing wrappers for all of the TH methods would be time-consuming and error prone. We need a better way to do this.

PyTorch cwrap

PyTorch implements its own cwrap tool to wrap the TH Tensor methods for use in the Python backend. We define a .cwrap file containing a series of C method declarations in our custom YAML format. The cwrap tool takes this file and outputs .cpp source files containing the wrapped methods in a format that is compatible with our THPTensor Python object and the Python C extension method calling format. This tool is used to generate code to wrap not only TH, but also CuDNN. It is defined to be extensible.

An example YAML “declaration” for the in-place addmv_ function is as follows:

```
[[
  name: addmv_
  cname: addmv
  return: self
  arguments:
    - THTensor* self
    - arg: real beta
      default: AS_REAL(1)
    - THTensor* self
    - arg: real alpha
      default: AS_REAL(1)
    - THTensor* mat
    - THTensor* vec
]]
```

The architecture of the cwrap tool is very simple. It reads in a file, and then processes it with a series of **plugins.** See tools/cwrap/plugins/__init__.py for documentation on all the ways a plugin can alter the code.

The source code generation occurs in a series of passes. First, the YAML “declaration” is parsed and processed. Then the source code is generated piece-by-piece - adding things like argument checks and extractions, defining the method header, and the actual call to the underlying library such as TH. Finally, the cwrap tool allows for processing the entire file at a time. The resulting output for addmv_ can be explored here.

In order to interface with the CPython backend, the tool generates an array of PyMethodDefs that can be stored or appended to the THPTensor’s tp_methods field.

In the specific case of wrapping Tensor methods, the build process first generates the output source file from TensorMethods.cwrap. This source file is #include‘d in the generic Tensor source file. This all occurs before the preprocessor does its magic. As a result, all of the method wrappers that are generated undergo the same pass as the THPTensor code above. Thus a single generic declaration and definition is specialized for each type as well.

Putting It All Together

So far, we have shown how we extend the Python interpreter to create a new extension module, how such a module defines our new THPTensor type, and how we can generate source code for Tensors of all types that interface with TH. Briefly, we will touch on compilation.

Setuptools allows us to define an Extension for compilation. The entire torch._C extension is compiled by collecting all of the source files, header files, libraries, etc. and creating a setuptools Extension. Then setuptools handles building the extension itself. I will explore the build process more in a subsequent post.

To summarize, let’s revisit our four questions:

- **How does PyTorch extend the Python interpreter to define a Tensor type that can be manipulated from Python code?**

It uses CPython’s framework for extending the Python interpreter and defining new types, while taking special care to generate code for all types.

- **How does PyTorch wrap the C libraries that actually define the Tensor’s properties and methods?**

It does so by defining a new type, THPTensor, that is backed by a TH Tensor. Function calls are forwarded to this tensor via the CPython backend’s conventions.

- **How does PyTorch cwrap work to generate code for Tensor methods?**

It takes our custom YAML-formatted code and generates source code for each method by processing it through a series of steps using a number of plugins.

- **How does PyTorch’s build system take all of these components to compile and generate a workable application?**

It takes a bunch of source/header files, libraries, and compilation directives to build an extension using Setuptools.

This is just a snapshot of parts of the build system for PyTorch. There is more nuance, and detail, but I hope this serves as a gentle introduction to a lot of the components of our Tensor library.

Resources:

- https://docs.python.org/3.7/extending/index.html is invaluable for understanding how to write C/C++ Extension to Python