TorchScript serialization

This document explains the TorchScript serialization format, and the anatomy of a call to `torch::jit::save()` or `torch::jit::load()` .

Overview

A serialized model (call it `model.pt`) is a ZIP archive containing many files. If you want to manually crack it open, you can call `unzip` on it to inspect the file structure directly:

```
$ unzip model.pt
Archive:  model.pt
  extracting ...

$ tree model/
├── code/
│   ├── __torch__.py
│   ├── __torch__.py.debug_pkl
│   ├── foo/
│   │   ├── bar.py
│   │   ├── bar.py.debug_pkl
├── data.pkl
├── constants.pkl
└── data/
    ├── 0
    └── 1
```

You'll notice that there are `.py` and `.pkl` files in this archive. That's because our serialization format tries to mimic Python's. All "code-like" information (methods, modules, classes, functions) are stored as human-readable `.py` containing valid Python syntax, and all "data-like" information (attributes, objects, etc.) are pickled using a subset of Python's pickle protocol.

A model is really a top-level module with some submodules, parameters, and so on depending on what the author needs. So, `data.pkl` contains the pickled top-level module. Deserializing the model is as simple as calling `unpickle()` on `data.pkl`, which will restore the module object state and load its associated code on demand.

Design Notes

Some things to keep in mind while working on the serialization code. These may help make technical decisions on which approach to take when making a change.

Do what Python does. When it comes to the serialized format, it's much simpler in the long-run to be consistent with whatever Python does. A good rule of thumb is: if I tried to interact with serialized artifacts using Python, would it work? i.e., all serialized code should be valid Python, and all pickled objects should be depickle-able by Python.

Being consistent with Python means our format is more debuggable (you can always crack it open and poke at it from Python) and leads to fewer surprises for developers familiar with Python but not familiar with TorchScript.

Human readable. In addition to being valid Python, serialized code should attempt to be readable Python. We should try to preserve the variable names that authors wrote, appropriately inline short expressions, and so on. This helps with debugging the serialized code.

No jitter. If we do:

```
m = MyModule()
m.save("foo.pt")
m_loaded = torch.load("foo.pt")
m_loaded.save("foo2.pt")
m_loaded2 = torch.load("foo2.pt")
```

We want the property that `m_loaded` and `m_loaded2` are identical. This "no-jitter" property is useful in catching bugs in the serialization process, and generally is desirable for debugging (models won't drift depending on how many times you saved/loaded them).

Initial load should be fast. Calling `load()` should be effectively instantaneous to a human. Anything that takes a long time (reading in tensor data, for example) should be done lazily.

`code/` : How code is serialized

At a high level, code serialization means:

1. Transforming `ClassType`s and `Function`s (called "code objects") into Python source code.
2. Placing the source code in the model ZIP archive.

Printing code objects as Python source

`PythonPrint` is the function that takes as input a `ClassType` or `Function` ("code object") and outputs Python source code. `ScriptModule`s are implemented as class types, so their methods and attributes will get serialized as well.

`PythonPrint` works by walking a `Graph` (the IR representation of either a `ClassType`'s method or raw `Function`) and emitting Python code that corresponds to it. The rules for emitting Python code are mostly straightforward and uninteresting. There are some extra pieces of information that `PythonPrint` tracks, however:

Class dependencies. While walking the graph, `PythonPrint` keeps track of what classes are used in the graph and adds them to a list of classes that the current code object depends on. For example, if we are printing a `Module`, it will depend on its submodules, as well as any classes used in its methods or attributes.

Uses of tensor constants. Most constants are inlined as literals, like strings or ints. But since tensors are potentially very large, when `PythonPrint` encouters a constant tensor it will emit a reference to a global `CONSTANTS` table (like `foo = CONSTANTS.c0`).

When importing, the importer will know how to resolve this reference into an actual tensor by looking it up in the tensor table. So `CONSTANTS.c0` means "this is the `0th` tensor in the tensor tuple in `constants.pkl` ." See the constants section for more info.

Original source range records. To aid debugging, `PythonPrint` remembers the "original" (user-written) location of the source code it's emitting. That way, when the user is debugging a model they loaded, they will see diagnostics that point to the code that they actually wrote, rather than the code that `PythonPrint` emitted.

The original source range records are pickled and saved in a corresponding `.debug_pkl` file with the same name as the code. You can think of this `.debug_pkl` file as a map between source ranges in the serialized code and the original user-written code.

Module information. Modules are special in a few ways. First are `Parameter` s: some module attributes are actually `Parameter` s, which have special properties (see the `torch.nn` documentation for exact details). We track which attributes are parameters by emitting a special assignment in the class body, like:

```
class MyModule(Module):
    __parameters__ = ["foo", "bar", ]
    foo : Tensor
    bar : Tensor
    attribute_but_not_param : Tensor
```

Another special thing with modules is that they are typically constructed in Python, and we do not compile the `__init__()` method. So in order to ensure they are statically typed, `PythonPrint` must enumerate a module's attributes (as you can see above), because it can't rely on compiling `__init__()` to infer the attributes.

A final special thing is that some modules (like `nn.Sequential`) have attributes that are not valid Python identifiers. We can't write

```
# wrong!
class MyModule(Module):
    0 : ASubmodule
    1 : BSubmodule
```

because this is not valid Python syntax (even though it is legal in Python to have attributes with those names!). So we use a trick where we write directly to the `__annotations__` dict:

```
class MyModule(Module):
    __annotations__ = []
    __annotations__["0"] = ASubmodule
    __annotations__["1"] = ASubmodule
```

Placing the source code in the archive

Once all code objects have been `PythonPrint` ed into source strings, we have to figure out where to actually put this source. Explaining this necessitates an introduction to `CompilationUnit` and `QualifiedName` . See the appendix on `CompilationUnit` for more info.

`CompilationUnit` : this is the owning container for all code objects associated with a given model. When we load, we load all the code objects to a single `CompilationUnit` .

`QualifiedName` : this is the fully qualified name for a code object. It is similar to qualified names in Python, and looks like `"foo.bar.baz"` . Each code object has a *unique* `QualifiedName` within a `CompilationUnit` .

The exporter uses the `QualifiedName` of a code object to determine its location in the `code/` folder. The way it does so is similar to how Python does it; for example, the class `Baz` with a `QualifiedName` `"foo.bar.Baz"` will be placed in `code/foo/bar.py` under the name `Baz` .

Classes at the root of the hierarchy are given the qualified name `__torch__` as a prefix, just so that they can go in `__torch__.py` . (Why not `__main__` ? Because pickle has weird special rules about things that live in `__main__`).

That's about it; there's some additional logic to make sure that within a file, we place the classes in reverse-dependency order so that we compile the "leaf" dependencies before things that depend on them.

How data is serialized

A model is really a top-level `ScriptModule` with any number of submodules, parameters, attributes, and so on. We implement a subset of the Pickle format necessary for pickling a module object.

`pickle` 's format was chosen due to:

- **user friendliness** - the attributes file can be loaded in Python with `pickle`
- **size limits** - formats such as Protobuf empose size limits on total message size, whereas pickle limits are on individual values (e.g. strings cannot be longer than 4 GB)
- **standard format** - `pickle` is a standard Python module with a reasonably simple format. The format is a program to be consumed by a stack machine that is detailed in Python's
- [`pickletools.py`]
- **built-in memoization** - for shared reference types (e.g. Tensor, string, lists, dicts)
- **self describing** - a separate definition file is not needed to understand the pickled data
- **eager mode save** - `torch.save()` already produces a `pickle` archive, so doing the same with attributes avoids introducing yet another format

`data.pkl` : How module object state is serialized

All data is written into the `data.pkl` file with the exception of tensors (see [the tensor section] below). "Data" means all parts of the module object state, like attributes, submodules, etc.

PyTorch functions defined in [torch/jit/_pickle.py] are used to mark special data types, such as this tensor table index or specialized lists.

`data/` : How tensors are serialized

During export a list of all the tensors in a model is created. Tensors can come from either module parameters or attributes of Tensor type.

Tensors are treated differently from other data (which is pickled using the standard pickling process) for a few reasons:

- Tensors regularly exceed the `pickle` file size limit.
- We'd like to be able to `mmap` Tensors directly.

- We'd like to maintain compatibility with regular `PyTorch` 's serialization format

`constants.pkl` : Constants in code

The `pickle` format enforces a separation between data and code, which the TorchScript serialization process represents by having `code/` and `data.pkl + tensors/` .

However, TorchScript inlines constants (i.e. `prim::Constant` nodes) directly into `code/` . This poses a problem for tensor constants, which are not easily representable in string form.

We can't put tensor constants in `data.pkl` , because the source code must be loaded *before* `data.pkl` , and so putting the tensor constants there would create a cyclic loading dependency.

We solve this problem by creating a separate `pickle` file called `constants.pkl` , which holds all tensor constants referenced in code. The load order will be explained in the next section.

`torch::jit::load()`

The load process has the following steps:

1. Unpickle `constants.pkl` , which produces a tuple of all tensor constants referenced in code.
2. Unpickle `data.pkl` into the top-level `Module` and return it.

The unpickling process consists of a single call to unpickle the module object contained in `data.pkl` . The `Unpickler` is given a callback that lets it resolved any qualified names it encounters into `ClassType` s. This is done by resolving the qualified name to the appropriate file in `code/` , then compiling that file and returning the appropriate `ClassType` .

This is why it's important to give code objects unique qualified names in the `CompilationUnit` . That way, every class that `Unpickler` encounters has a deterministic location in `code/` where it is stored.

`Unpickler` is also responsible for resolving references to tensors into actual `at::Tensor` s. This is done by looking up offsets in the tensor table during the unpickling process, (soon to be replaced with the same pickling strategy as all other data).

`__getstate__` and `__setstate__`

Like in Python's `pickle` , users can customize the pickling behavior of their class or module by implementing `__getstate__()` and `__setstate__()` methods. For basic usage, refer to the relevant [Python docs].

Calls to `__getstate__` and `__setstate__` are handled transparently by `Pickler` and `Unpickler` , so the serialization process shouldn't worry about it too much.

One thing worth calling out is that the compiler implements a few special type inference behaviors to cheat the fact that users currently cannot type annotate `Module` s.

`__getstate__` and `__setstate__` do not require type annotations. For `__getstate__` , the compiler can fully infer the return based on what attributes the user is returning. Then, `__setstate__` simply looks up the return type of `__getstate__` and uses that as its input type.

For example:

```
class M(torch.nn.Module):
    def __init__(self):
        self.a = torch.rand(2, 3)
        self.b = torch.nn.Linear(10, 10)

    def __getstate__(self):
        # Compiler infers that this is a tuple of (Tensor, Linear)
        return (self.a, self.b)

    def __setstate__(self, state):
        # Don't need to annotate this, we know what type `state` is!
        self.a = state[0]
        self.b = state[1]
```

Appendix: `CompilationUnit` and code object ownership

`CompilationUnit` performs two functions:

1. It is the owner (in a C++ sense) for all code objects.
2. It forms a namespace in which code objects must have unique names.

A `CompilationUnit` is created whenever `torch::jit::load()` is invoked, to place the newly deserialized code objects in. In Python, there is a single global `CompilationUnit` that holds all code objects defined in Python.

`CompilationUnit` ownership semantics

There are a few different entities that participate in the ownership model: **`CompilationUnit`** : A container that owns code objects and gives them name. Every code object has a unique qualified name within the CompilationUnit.

There are two kinds of code objects: `Function` s and `ClassType` s. **`Function`** : A `Graph` with an associated executor. The `Graph` may own `ClassType` s, since some `Value` s hold a `shared_ptr` to their type (for now). The `Graph` may also weakly reference other `Function` s through function calls.

`ClassType` : A definition of a type. This could refer to a user-defined TorchScript class, or a `ScriptModule` . Owns other its attribute types (including other ClassTypes). Weakly references the class's methods (`Function` s).

`Object` : An instance of a particular class. Own the `CompilationUnit` that owns its `ClassType` . This is to ensure that if the user passes the object around in C++, all its code will stay around and methods will be invokable.

`Module` : A view over a `ClassType` and the `Object` that holds its state. Also responsible for turning unqualified names (e.g. `forward()`) into qualified ones for lookup in the owning `CompilationUnit` (e.g. `__torch__.MyModule.forward`). Owns the `Object` , which transitively owns the `CompilationUnit` .

`Method` : A tuple of `(Module, Function)` .

Code object naming

`CompilationUnit` maintains a namespace in which all code objects (`ClassType` s and `Function` s) are uniquely named. These names don't have any particular meaning, except that they uniquely identify a code object during serialization and deserialization. The basic naming scheme is:

- Everything starts in the `__torch__` namespace.

- Classes are named parallel to Python's module namespacing: so class `Bar` in `foo.py` would become `__torch__.foo.Bar`.
- Methods are attached to the module's namespace. So `Bar.forward()` would be `__torch__.foo.Bar.forward`.

There are some caveats:

Some `CompilationUnit`s have no prefix: For testing and other internal purposes, occasionally it's useful to have no prefixes on names. In this case, everything is just a bare name inside the `CompilationUnit`. Users cannot construct `CompilationUnits that look like this.

Name mangling: In Python, we can construct code objects that have the same qualified name. There are two cases where this happens:

1. For `ScriptModule`s, since every `ScriptModule` is a singleton class in the JIT, a user that is constructing multiple `ScriptModule`s will create multiple corresponding `ClassType`s with identical names.
2. Nesting functions will also cause qualified name clashes, due to limitations in Python. In these cases, we mangle the names of the code objects before they are placed in the global Python `CompilationUnit`.

The rules for mangling are simple. Say we have a qualified name `__torch__.foo.Bar`:

```
__torch__.foo.Bar                    # first time, unchanged
__torch__.foo.__torch_mangle_0.Bar   # second time, when we request a mangle
__torch__.foo.__torch_mangle_1.Bar   # and so on
```

Notice that we mangle the namespace before `Bar`. This is so that when we pretty-print code, the unqualified name (`Bar`) is unchanged. This is a useful property so that things like trace-checking are oblivious to the mangling.