Saving Non-Tensor Parameters in PyTorch

Bilin Sun

March 2024

1 Introduction

In the process of training neural network models, the ability to save parameters to a checkpoint file allows great flexibility: by saving to checkpoints periodically, we can make the training process robust against interruptions. For instance, the UChicago AI cluster enforces a timeout limit of four hours. If a task takes longer than that time, it simply gets killed. Having checkpoints then guarantees that we do not lose our progress.

In this report, we document the challenges when attempting to save non-tensor parameters with PyTorch. We review some existing methods to save them, alongside our new method of using *hooks*, a feature that's relatively new and rarely covered in existing online documentations.

2 PyTorch model saving overview

PyTorch models are created by deriving the `nn.Module` class. The class definition of each individual layer inherits `nn.Module`. From them, we build higher-level structures that are themselves `nn.Module`s. This goes all the way up to the full model. Therefore, a model can be viewed as a nested structure of `nn.Module`s.[1]

`nn.Module` provides a method `state_dict()` that, to borrow the official documentation's words, *returns a dictionary containing references to the whole state of the module.* As this method effectively serves as a getter, for the rest

[1]Given an `nn.Module`, we then call the contained subordinate `nn.Module`s *submodules*. To address a potential confusion of terms, as the "sub-" in "submodules" carries a very different meaning from the "sub-" in "subclasses", we avoid the term "subclasses" altogether, and instead use "derived classes" or "child classes".

of the report, we use `state_dict` to denote the dictionary returned by this method, and `state_dict()` to denote the method itself.[2]

For usual practical purposes, all that `state_dict` includes suffices for loading the model. Hence, the common practice for saving checkpoints is to simply `torch.save()` the `state_dict` to a checkpoint file.

Under the hood, `state_dict()` retrieves all the *parameters* and *buffers* of the current `nn.Module` on-the-fly, and makes recursive calls for all the submodules. Among the attributes of an `nn.Module` (or any derived class thereof), all tensor parameters declared as `nn.Parameter`s are automatically registered as *parameters*, and so are included in the `state_dict`, which is the reason that `state_dict()` works out-of-the-box for most people. On the other hand, if we make use of "raw" tensor attributes in the `nn.Module` and want to save them, then we need to manually call `register_buffer()` to, as the name aptly suggests, register the tensor as a *buffer*. As outlined in the official documentation, when the second argument `val` to `register_buffer('name', val)` is a tensor, `register_buffer`

- declares and assigns an attribute of `self` with the name `'name'` if it's previously undeclared, as if you wrote `self.name = val`; and
- registers this name to the list of *buffers* that will be in turn considered by `state_dict()`.

However, if `val` is `None`, then `register_buffer` takes neither of these actions, as if you did not make he call to `register_buffer` at all.

Here is a MWE[3], where we skip actually writing to a file to keep it simple.

```python
import numpy, torch

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.register_buffer('x', torch.FloatTensor
            ([1,2,3]))
```

[2]That is, we pretend this method has the `@property` decorator, so as to give a name to the dictionary. The PyTorch official tutorials also seem to use such nomenclature. However, it is important to know that the PyTorch source does not in fact have such an attribute, and using `state_dict` without parentheses is **not** supported.

[3]Adapted from https://discuss.pytorch.org/t/initialization-of-register-buffer-for-loading-model/123990/4

```python

model = Model()
model.x[0] = 0
checkpoint = model.state_dict()

model_loaded = Model()
model_loaded.load_state_dict(checkpoint)
print(f"Model loaded successfully from dictionary.")
print(f"model_loaded.x = {model_loaded.x}")
```

As expected, the output is as follows.

```
Model loaded successfully from dictionary.
model_loaded.x = tensor([0., 2., 3.])
```

3 The challenge of a lazy-initialized tensor

In the project that I am working on, a tensor named `p` is initialized to `None`. Not even its shape can be easily determined at the beginning, and it's only assigned a tensor value after a number of epochs, specifically 350, have elapsed.

Once assigned, it needs to remain fixed over the whole training process, so we should save it to and recover it from checkpoints. Here's a toy formulation:

```python
import torch

class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        # code to register p
        # ...
        if epoch > 350:
            self.p = torch.FloatTensor([1,2,3])

model = Model(epoch = 351)
checkpoint = model.state_dict()

```

```
model_loaded = Model()
model_loaded.load_state_dict(checkpoint)
model_loaded.epoch = 351
```

Our goal is to apply `register_buffer` to `p` so that it gets saved. Recall that it does not help to `self.register_buffer('p', None)` before `p` is assigned, as such one line is effectively a no-op.

A natural idea is:

```
class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        if epoch > 350:
            self.register_buffer('p', torch.
               FloatTensor([1,2,3]))
```

which works correctly to *save* `p` as needed. However, when `model_loaded` gets *loaded*, we will get

```
RuntimeError: Error(s) in loading state_dict for Model
   :
    Unexpected key(s) in state_dict: "p".
```

The error seems to suggest that everything to be loaded has to be declared, so we may try adding one line to `__init__`:

```
class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        self.p = None # added line
        if epoch > 350:
            self.register_buffer('p', torch.
               FloatTensor([1,2,3]))
```

The consequent error is:

```
KeyError: "attribute 'p' already exists"
```

This issue has apparently stumbled other people[4]. Apparently we can make a one-liner fix:

```
class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        if epoch > 350: # added conditional
            self.register_buffer('p', torch.
               FloatTensor([1,2,3]))
        else:
            self.p = None
```

However, the attribute-exists error persists, suggesting that `p` needs to be "declared" in a even stronger way that's recognizable by PyTorch, presumably using `register_buffer`. Recall that we did not know the tensor shape beforehand; all that we can reasonably put is a dummy tensor:

```
class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        if epoch > 350:
            self.register_buffer('p', torch.
               FloatTensor([1,2,3]))
        else:
            self.register_buffer('p', torch.
               FloatTensor([0]))
```

Still, this does not work:

```
runtimeerror: error(s) in loading state_dict for model
   :
    size mismatch for p: copying a param with shape
       torch.size([3]) from checkpoint, the shape in
       current model is torch.size([1]).
```

The only solution, then, seems to be to just cheat, and pretending that we do know the tensor shape:

[4] See `https://github.com/pytorch/pytorch/issues/3232`

```python
class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        if epoch > 350:
            self.register_buffer('p', torch.
                FloatTensor([1,2,3]))
        else:
            self.register_buffer('p', torch.
                FloatTensor([0,0,0]))
```

This allows the program to finish correctly, and if we print out `model_loaded.p` in the end, we can observe that the values are correctly loaded. But this defeats my intention: I don't know the dimensions when the model is being initialized, and worst yet, some parts of the code may depend on `p` being `None` rather than a tensor to know that `p` isn't assigned actual values yet. Having to assign `p` this early may force refactoring code elsewhere.

4 Hooks to our aid

At this point, all simple manipulations of `register_buffer()` have been exhausted, and we have to look into more non-trivial tricks. As there's no single way to frame such a problem, searching up a solution is difficult, and having tried various combinations of keywords, we conclude that no exisiting online documentation, blog or discussion post has addressed the very same issue.

There's an official guide[5] on saving and loading models, but it's for beginners and does not address such a nuanced case. Some PyTorch discussion posts[6] sketch a possible implementation: make a wrapper like `load_my_state_dict()` around `load_state_dict()` that handles `p` separately and lets `load_state_dict()` do what it's able to handle. However, this would force an even worse refactoring since we will need to replace every occurrence of `load_state_dict()` with `load_my_state_dict()` in all the source code. It's also possible to force

[5]Refer to `https://pytorch.org/tutorials/beginner/saving_loading_models.html`

[6]See `https://webcache.googleusercontent.com/search?q=cache:RiofqncoBysJ:https://discuss.pytorch.org/t/how-to-load-part-of-pre-trained-model/1113&hl=en&gl=us`, `https://discuss.pytorch.org/t/how-to-load-part-of-pre-trained-model/1113/2`, and `https://stackoverflow.com/questions/53907073/problem-with-missing-and-unexpected-keys-while-loading-my-model-in-pytorch`

PyTorch to neglect *all* discrepancies[7] and force load a `state_dict` that has different keys than in the current module. I have yet to check whether it allows the loading of "undeclared" attributes like our `p`, but even if it does, this easy route may be hazardous too: forcing open up the firewall will forever let any new bug in checkpoint saving go undetected.

With quite some search I was eventually able to find the issue taken care of by people who were trying to save non-tensor objects[8]. The *hook* methods proposed in the post, `set_extra_state()` and `get_extra_state()`, are thankfully now implemented in PyTorch. The two methods are however too new to appear in any tutorial online, so we have to fall back to the documentation for `set_extra_state()` [9] and `get_extra_state()` [10] which to be honest does not explain the usage very well. The definition in the source code provides not much information either, since the two methods are supposed to be implemented (i.e. overwritten) by the user. Only by searching for other occurrences of these method names in the PyTorch source did I grasp the usage, listed as follows.

- The PyTorch user implements (overwrites) `set_extra_state()`, which takes one parameter and does not return, and `get_extra_state()`, which takes no parameter and can return `Any` type of value.
- As `state_dict()` runs, it calls `get_extra_state()` (if implemented), and stores the return value of `get_extra_state()` in the state dictionary about to be returned: `destination[extra_state_key] = self.get_extra_state()`. As of this write-up, `extra_state_key` is defined to be an optional prefix (exactly the optional parameters [11] you can pass to `state_dict()`) concatenated with `'_extra_state'` [12]. The recursions in `state_dict` accumulate in the prefix, so that the key name for the `extra_state_dict` of a specific module depends on its location in the model's `nn.Module`

[7]See https://stackoverflow.com/questions/63057468/how-to-ignore-and-initialize-missing-keys-in-state-dict

[8]See https://github.com/pytorch/pytorch/issues/62094

[9]Refer to https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.set_extra_state

[10]Refer to https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.get_extra_state

[11]Refer to https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.state_dict

[12]Refer to, e.g. https://github.com/pytorch/pytorch/blob/27389e03f03b7dbbb69f93d6889152923cc6d4da/torch/nn/modules/module.py#L1822.

hierarchy.

- The user saves the return value, the modified `state_dict`, to a checkpoint.
- As `load_state_dict()` runs, it calls `set_extra_state()` (if implemented) before returning, and passes what `get_extra_state()` returned earlier as the only argument to `set_extra_state()`.

These hooks allow us to come up with a satisfactory working implementation. Here is a full MWE:

```python
import torch

class Model(torch.nn.Module):
    def __init__(self, epoch = 0):
        super(Model, self).__init__()
        self.epoch = epoch
        self.p = None
        if epoch > 350:
            self.p = torch.FloatTensor([1,2,3])
    def get_extra_state(self):
        extra_state_dict = {'p': self.p}
        return extra_state_dict
    def set_extra_state(self, extra_state_dict):
        self.p = extra_state_dict['p']

model = Model(epoch = 351)
checkpoint = model.state_dict()

model_loaded = Model()
model_loaded.load_state_dict(checkpoint)
print(f"Model loaded successfully from dictionary.")
print(f"model_loaded.p = {model_loaded.p}")
```

Output:

```
Model loaded successfully from dictionary.
model_loaded.p = tensor([1., 2., 3.])
```

For better extensibility, we can rewrite the two hooks as:

```python
def get_extra_state(self):
    extra_state_dict = {}
    extra_state_strs = [
        'p',
    ]
    for extra_state_str in extra_state_strs:
        if hasattr(self, extra_state_str):
            extra_state_dict[extra_state_str] =
                getattr(self, extra_state_str)
    return extra_state_dict

def set_extra_state(self, extra_state_dict):
    for key, val in extra_state_dict.items():
        setattr(self, key, val)
```

so that we can simply add more entries to `extra_state_strs` to specify what extra attributes to save.

We observe that being able to save `p` has indeed eliminated the drop in inference accuracy (visible in figure 1).

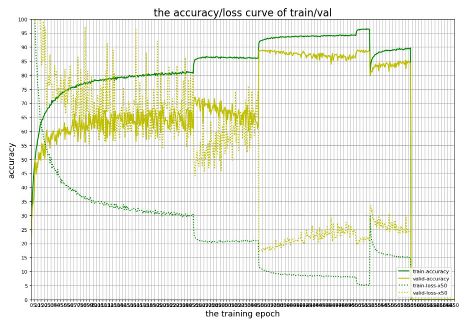

Figure 1: Before the fix

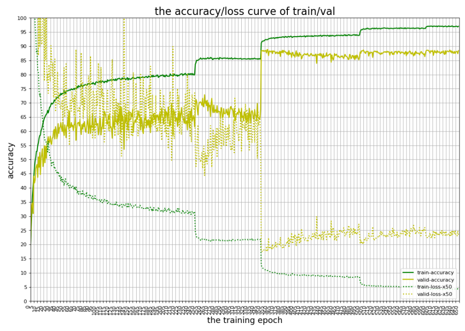

Figure 2: After the fix

5 Adding backwards compatibility

Before adding the ability to save `p`, I trained the model without saving the `p` values. As a consequence, whenever training tasks using the unmodified code was interrupted and forced to reload from a checkpoint, there was a mysterious drop in inference accuracy. After modifying code to correctly save `p`, the drop was eliminated.

My model is such that, if at some epoch after the 350th the model finds `p` to be `None`, then it re-generates generates a value for `p`, which is bound to be different from the previously generated value, as the necessary information has mutated. If, therefore, `p` is not genuinely uninitialized but rather lost during saving to and loading from a checkpoint, then training results will be undesired, and all subsequent checkpoints useless.

However, historical checkpoints for epochs <350 can be reused for the corrected model, as `p` is irrelevant in the that first stage. Therefore, we would like to utilize these checkpoints to skip recomputation and save time. A simplified formulation is as follows:

```python
import torch

class Model_old(torch.nn.Module):
    def __init__(self):
        super(Model_old, self).__init__()
        self.p = None

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.p = None
    def set_extra_state(self, extra_state_dict):
        if "p" in extra_state_dict:
            self.p = extra_state_dict['p']
        else:
            self.p = None

model = Model_old()
checkpoint = model.state_dict()

model_loaded = Model()
model_loaded.load_state_dict(checkpoint)
print(f"Model loaded successfully from dictionary.")
print(f"model_loaded.p = {model_loaded.p}")
```

As intuitive such code is, it does not work, and produces the following error:

```
RuntimeError: Error(s) in loading state_dict for
   Model2:
    Missing key(s) in state_dict: "_extra_state".
```

The message implies that the "`extra_state_dict`"[13] item of the `state_dict` is missing altogether. While the `set_extra_state()` we furnish does handle the case where `extra_state_dict` is empty, PyTorch bugs out before calling `set_extra_state()`.

We have to inject `extra_state_dict` items into the checkpoint before the function call to `set_extra_state()` within `load_state_dict()`. A naive implementation would be to add entries to the loaded `state_dict` before `load_state_dict()`:

```
checkpoint["_extra_state"] = {}
model_loaded.load_state_dict(checkpoint)
```

However, when the model is non-trivial with nested modules, this process will be recursive and complicated for reasons explained previously: the actual key name for the `extra_state_dict` of each submodule has to be probed in the model's `nn.Module` hierarchy, so the key will be something like `module.layers.0.conv2._extra_state` in practice. It would easier to perform the injection with more context.A

The modern PyTorch infrastructure has another hook that aptly fits this need: we can register a hook that will be run early on by `load_state_dict()`, in particular before `set_extra_dict()`, using an internal undocumented method `_register_load_state_dict_pre_hook()` of `nn.Module`. It accepts one function `f` of the same module for argument, where `f` take seven arguments[14] `state_dict`, `prefix`, `local_metadata`, `strict`, `missing_keys`, `unexpected_keys`, and `error_msgs`. Cross compare these parameters with those for publicly available and documented hooks[15], it's not too difficult to figure out what each parameter stands for. Notable among the parameters are `state_dict`, which is guaranteed to be a "reference" of the module's (whereby modifications take effects beyond the scope of the function as dicitonaries are

[13]This is another convenience name that we make up.

[14]Refer to https://github.com/pytorch/pytorch/blob/27389e03f03b7dbbb69f93d68891 52923cc6d4da/torch/nn/modules/module.py#L2006 to see the implementation.

[15]E.g. `register_state_dict_pre_hook()`. Refer to the official documentations.

mutable) and `prefix`, which when the function is called for each submodule gives the correct prefix based on the position in the model hierarchy. This allows for a simple implementation:

```python
def add_extra_state_dict(state_dict, prefix,
        local_metadata, strict, missing_keys,
            unexpected_keys, error_msgs):
    extra_state_key = prefix + '_extra_state'
    if extra_state_key not in state_dict:
        state_dict[extra_state_key] = dict()

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.p = None
        self._register_load_state_dict_pre_hook(
            add_extra_state_dict)
    def set_extra_state(self, extra_state_dict):
        if "p" in extra_state_dict:
            self.p = extra_state_dict['p']
        else:
            self.p = None
```

6 Conclusion

This technical report summarizes two non-canonical demands for checkpoint saving and loading, namely saving lazy-initialized tensors and loading a checkpoint that's only lacking `extra_state` dictionaries, and solutions to them. The solutions make use of hooks that are, very new or private, not been extensively covered in documentations and tutorials. This technical report, in addition, documents a few common error messages stemming from incorrect usage of PyTorch checkpoint-saving facilities, and can therefore serve as a reference point when one meets the same errors.