Table of Contents

Colab Notebook GitHub

Saving and Loading Models

Author: Matthew Inkawhich

This document provides solutions to a variety of use cases regarding the saving and loading of PyTorch models. Feel free to read the whole document, or just skip to the code you need for a desired use case.

When it comes to saving and loading models, there are three core functions to be familiar with:

1. torch.save: Saves a serialized object to disk. This function uses Python's pickle utility for serialization. Models, tensors, and dictionaries of all kinds of objects can be saved using this function.
2. torch.load: Uses pickle's unpickling facilities to deserialize pickled object files to memory. This function also facilitates the device to load the data into (see Saving & Loading Model Across Devices).
3. torch.nn.Module.load_state_dict: Loads a model's parameter dictionary using a deserialized *state_dict*. For more information on *state_dict*, see What is a state_dict?.

Contents:

What is a `state_dict`?

In PyTorch, the learnable parameters (i.e. weights and biases) of an `torch.nn.Module` model are contained in the model's *parameters* (accessed with `model.parameters()`). A *state_dict* is simply a Python dictionary object that maps each layer to its parameter tensor. Note that only layers with learnable parameters (convolutional layers, linear layers, etc.) and registered buffers (batchnorm's running_mean) have entries in the model's *state_dict*. Optimizer objects (`torch.optim`) also have a *state_dict*, which contains information about the optimizer's state, as well as the hyperparameters used.

Because *state_dict* objects are Python dictionaries, they can be easily saved, updated, altered, and restored, adding a great deal of modularity to PyTorch models and optimizers.

Example:

Let's take a look at the *state_dict* from the simple model used in the Training a classifier tutorial.

```
# Define model
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initialize model
model = TheModelClass()

# Initialize optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])
```

Output:

```
Model's state_dict:
conv1.weight     torch.Size([6, 3, 5, 5])
conv1.bias   torch.Size([6])
conv2.weight     torch.Size([16, 6, 5, 5])
conv2.bias   torch.Size([16])
fc1.weight   torch.Size([120, 400])
fc1.bias     torch.Size([120])
fc2.weight   torch.Size([84, 120])
fc2.bias     torch.Size([84])
fc3.weight   torch.Size([10, 84])
fc3.bias     torch.Size([10])

Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False, 'params': [4675713712, 4675713784,
4675714000, 4675714072, 4675714216, 4675714288, 4675714432, 4675714504, 4675714648, 4675714720]}]
```

Saving & Loading Model for Inference

Save/Load `state_dict` (Recommended)

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

• NOTE

The 1.6 release of PyTorch switched `torch.save` to use a new zip file-based format. `torch.load` still retains the ability to load files in the old format. If for any reason you want `torch.save` to use the old format, pass the `kwarg``parameter ``_use_new_zipfile_serialization=False`.

When saving a model for inference, it is only necessary to save the trained model's learned parameters. Saving the model's *state_dict* with the `torch.save()` function will give you the most flexibility for restoring the model later, which is why it is the recommended method for saving models.

A common PyTorch convention is to save models using either a `.pt` or `.pth` file extension.

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results.

• NOTE

Notice that the `load_state_dict()` function takes a dictionary object, NOT a path to a saved object. This means that you must deserialize the saved *state_dict* before you pass it to the `load_state_dict()` function. For example, you CANNOT load using `model.load_state_dict(PATH)`.

• NOTE

If you only plan to keep the best performing model (according to the acquired validation loss), don't forget that `best_model_state = model.state_dict()` returns a reference to the state and not its copy! You must serialize `best_model_state` or use `best_model_state = deepcopy(model.state_dict())` otherwise your best `best_model_state` will keep getting updated by the subsequent training iterations. As a result, the final model state will be the state of the overfitted model.

Save/Load Entire Model

Save:

```
torch.save(model, PATH)
```

Load:

```
# Model class must be defined somewhere
model = torch.load(PATH)
model.eval()
```

This save/load process uses the most intuitive syntax and involves the least amount of code. Saving a model in this way will save the entire module using Python's pickle module. The disadvantage of this approach is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved. The reason for this is because pickle does not save the model class itself. Rather, it saves a path to the file containing the class, which is used during load time. Because of this, your code can break in various ways when used in other projects or after refactors.

A common PyTorch convention is to save models using either a `.pt` or `.pth` file extension.

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results.

Export/Load Model in TorchScript Format

One common way to do inference with a trained model is to use TorchScript, an intermediate representation of a PyTorch model that can be run in Python as well as in a high performance environment like C++. TorchScript is actually the recommended model format for scaled inference and deployment.

• NOTE

Using the TorchScript format, you will be able to load the exported model and run inference without defining the model class.

Export:

```
model_scripted = torch.jit.script(model) # Export to TorchScript
model_scripted.save('model_scripted.pt') # Save
```

Load:

```
model = torch.jit.load('model_scripted.pt')
model.eval()
```

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results.

For more information on TorchScript, feel free to visit the dedicated tutorials. You will get familiar with the tracing conversion and learn how to run a TorchScript module in a C++ environment.

Saving & Loading a General Checkpoint for Inference and/or Resuming Training

Save:

```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
optimizer = TheOptimizerClass(*args, **kwargs)

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

model.eval()
# - or -
model.train()
```

When saving a general checkpoint, to be used for either inference or resuming training, you must save more than just the model's *state_dict*. It is important to also save the optimizer's *state_dict*, as this contains buffers and parameters that are updated as the model trains. Other items that you may want to save are the epoch you left off on, the latest recorded training loss, external `torch.nn.Embedding` layers, etc. As a result, such a checkpoint is often 2~3 times larger than the model alone.

To save multiple components, organize them in a dictionary and use `torch.save()` to serialize the dictionary. A common PyTorch convention is to save these checkpoints using the `.tar` file extension.

To load the items, first initialize the model and optimizer, then load the dictionary locally using `torch.load()`. From here, you can easily access the saved items by simply querying the dictionary as you would expect.

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results. If you wish to resuming training, call `model.train()` to ensure these layers are in training mode.

Saving Multiple Models in One File

Save:

```
torch.save({
            'modelA_state_dict': modelA.state_dict(),
            'modelB_state_dict': modelB.state_dict(),
            'optimizerA_state_dict': optimizerA.state_dict(),
            'optimizerB_state_dict': optimizerB.state_dict(),
            ...
            }, PATH)
```

Load:

```
modelA = TheModelAClass(*args, **kwargs)
modelB = TheModelBClass(*args, **kwargs)
optimizerA = TheOptimizerAClass(*args, **kwargs)
optimizerB = TheOptimizerBClass(*args, **kwargs)

checkpoint = torch.load(PATH)
modelA.load_state_dict(checkpoint['modelA_state_dict'])
modelB.load_state_dict(checkpoint['modelB_state_dict'])
optimizerA.load_state_dict(checkpoint['optimizerA_state_dict'])
optimizerB.load_state_dict(checkpoint['optimizerB_state_dict'])

modelA.eval()
modelB.eval()
# - or -
modelA.train()
modelB.train()
```

When saving a model comprised of multiple `torch.nn.Modules`, such as a GAN, a sequence-to-sequence model, or an ensemble of models, you follow the same approach as when you are saving a general checkpoint. In other words, save a dictionary of each model's *state_dict* and corresponding optimizer. As mentioned before, you can save any other items that may aid you in resuming training by simply appending them to the dictionary.

A common PyTorch convention is to save these checkpoints using the `.tar` file extension.

To load the models, first initialize the models and optimizers, then load the dictionary locally using `torch.load()`. From here, you can easily access the saved items by simply querying the dictionary as you would expect.

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results. If you wish to resuming training, call `model.train()` to set these layers to training mode.

Warmstarting Model Using Parameters from a Different Model

Save:

```
torch.save(modelA.state_dict(), PATH)
```

Load:

```
modelB = TheModelBClass(*args, **kwargs)
modelB.load_state_dict(torch.load(PATH), strict=False)
```

Partially loading a model or loading a partial model are common scenarios when transfer learning or training a new complex model. Leveraging trained parameters, even if only a few are usable, will help to warmstart the training process and hopefully help your model converge much faster than training from scratch.

Whether you are loading from a partial *state_dict*, which is missing some keys, or loading a *state_dict* with more keys than the model that you are loading into, you can set the `strict` argument to **False** in the `load_state_dict()` function to ignore non-matching keys.

If you want to load parameters from one layer to another, but some keys do not match, simply change the name of the parameter keys in the *state_dict* that you are loading to match the keys in the model that you are loading into.

Saving & Loading Model Across Devices

Save on GPU, Load on CPU

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
device = torch.device('cpu')
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location=device))
```

When loading a model on a CPU that was trained with a GPU, pass `torch.device('cpu')` to the `map_location` argument in the `torch.load()` function. In this case, the storages underlying the tensors are dynamically remapped to the CPU device using the `map_location` argument.

Save on GPU, Load on GPU

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
device = torch.device("cuda")
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.to(device)
# Make sure to call input = input.to(device) on any input tensors that you feed to the model
```

When loading a model on a GPU that was trained and saved on GPU, simply convert the initialized `model` to a CUDA optimized model using `model.to(torch.device('cuda'))`. Also, be sure to use the `.to(torch.device('cuda'))` function on all model inputs to prepare the data for the model. Note that calling `my_tensor.to(device)` returns a new copy of `my_tensor` on GPU. It does NOT overwrite `my_tensor`. Therefore, remember to manually overwrite tensors: `my_tensor = my_tensor.to(torch.device('cuda'))`.

Save on CPU, Load on GPU

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
device = torch.device("cuda")
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location="cuda:0"))  # Choose whatever GPU device number you want
model.to(device)
# Make sure to call input = input.to(device) on any input tensors that you feed to the model
```

When loading a model on a GPU that was trained and saved on CPU, set the `map_location` argument in the `torch.load()` function to `cuda:device_id`. This loads the model to a given GPU device. Next, be sure to call `model.to(torch.device('cuda'))` to convert the model's parameter tensors to CUDA tensors. Finally, be sure to use the `.to(torch.device('cuda'))` function on all model inputs to prepare the data for the CUDA optimized model. Note that calling `my_tensor.to(device)` returns a new copy of `my_tensor` on GPU. It does NOT overwrite `my_tensor`. Therefore, remember to manually overwrite tensors: `my_tensor = my_tensor.to(torch.device('cuda'))`.

Saving `torch.nn.DataParallel` Models

Save:

```
torch.save(model.module.state_dict(), PATH)
```

Load:

```
# Load to whatever device you want
```

`torch.nn.DataParallel` is a model wrapper that enables parallel GPU utilization. To save a `DataParallel` model generically, save the `model.module.state_dict()`. This way, you have the flexibility to load the model any way you want to any device you want.

Total running time of the script: (0 minutes 0.000 seconds)

Rate this Tutorial 

Docs

Access comprehensive developer documentation for PyTorch

View Docs

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials

Resources

Find development resources and get your questions answered

View Resources