Lab5B - Saving and Loading Models

In the process of training the model, you may stop the training temporarily and resume it later. You may also want to save the best model which may not be the model generated in the last iteration. More importantly, after completion of training, you want to deploy your model to the field. All this requires you to save and load the model.

Objectives:

In this practical, students learn how to:

1. Save and loading models
2. Resume previous training

References:

1. [Saving and loading models (https://pytorch.org/tutorials/beginner/saving_loading_models.html)](https://pytorch.org/tutorials/beginner/saving_loading_models.html)

In [1]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=True).
```

In [2]:
```
cd "./gdrive/MyDrive/UCCD3074_Labs/UCCD3074_Lab5"
```

```
/content/gdrive/MyDrive/UCCD3074_Labs/UCCD3074_Lab5
```

Import the required library.

In [3]:
```
# imports
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F

import torchvision
import torchvision.transforms as transforms

from torch.utils.data import DataLoader

import torch.optim as optim
import os

from cifar10 import CIFAR10
```

In [4]:
```
if not os.path.exists("./models"):
    os.mkdir("models")
```

1. Introduction

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.

What is a `state_dict()` ?

- Each model has a `state_dict` . The model state_dict is simply a Python dictionary object that maps each layer to its parameter tensors stored in `model.parameters()` . `state_dict` stores the following tensors:
  - learnable parameters (convolutional layers, linear layers, etc.)
  - registered buffers (batchnorm's running mean).
- The optimizer object (`torch.optim`) also have a `state_dict` , which contains information about
  - the optimizer's state
  - 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.

First, let's build our model.

In [5]:
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(8)

        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn2   = nn.BatchNorm2d(16)

        self.fc1 = nn.Linear(16*30*30, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)

        x = x.view(x.size(0), -1) # flat
        x = self.fc1(x)
        x = self.fc2(x)

        return x
```

The following shows the `state_dict` of the model. Note that `state_dict` stores not only the *parameters* (weight and bias) of the trainable layers but also the *running mean* of the batch norm layer.

In [6]:
```
model = Net()
```

In [7]:
```
# 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())
```

```
Model's state_dict:
conv1.weight     torch.Size([8, 3, 3, 3])
conv1.bias       torch.Size([8])
bn1.weight       torch.Size([8])
bn1.bias         torch.Size([8])
bn1.running_mean         torch.Size([8])
bn1.running_var          torch.Size([8])
bn1.num_batches_tracked          torch.Size([])
conv2.weight     torch.Size([16, 8, 3, 3])
conv2.bias       torch.Size([16])
bn2.weight       torch.Size([16])
bn2.bias         torch.Size([16])
bn2.running_mean         torch.Size([16])
bn2.running_var          torch.Size([16])
bn2.num_batches_tracked          torch.Size([])
fc1.weight       torch.Size([256, 14400])
fc1.bias         torch.Size([256])
fc2.weight       torch.Size([10, 256])
fc2.bias         torch.Size([10])
```

The following code shows the `state_dict` of the optimizer. It stores the *hyperparameter* settings (e.g., `lr` , `momentum` , `dampening` , `weight_decay` , `nesterov`) as well as the *optimizer* states (`params`)

In [8]:
```
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
```

In [9]:
```
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])
```

```
Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.1, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False, 'params': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]}]
```

1.1 Saving & Loading Model Parameters Only

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

When saving a model for inference, it is only necessary to save the trained model's learned parameters. We do not need to save the network structure itself. To do that, use the command `torch.save()` . A common PyTorch convention is to save models using either a `.pt` or `.pth` file extension.

In [10]:
```
model = Net()
torch.save(model.state_dict(), "./models/saved_params.pt")
```

model. **`load_state_dict(torch.load(PATH))`**

To load the model parameters, use the model's function `load_state_dict()` . `load_state_dict()` takes a dictionary object, NOT a path to a saved object. So, you must deserialize the saved state_dict first (`torch.load(PATH)`) before you pass it to the `load_state_dict()` function.

In [11]:
```
new_model = Net()
_ = new_model.load_state_dict(torch.load("./models/saved_params.pt"))
```

1.2 Saving the Entire Model

The previous method only saves the model *parameters* but not the *network* itself. As a result, the saved parameters must be accompanied by the *model class*, i.e., the class `Net` , so that we can create the *network* first before loading the parameters. Because of this, your code can break in various ways when used in other projects or after refactors.

`torch.save(model, PATH)`

You may save the whole model and use it for inference by providing `model` rather than `model.state_dict()` as the argument for `torch.save` . This eliminates the need to attach the model class together with your saved model file.

In [12]:
```
model = Net()
torch.save(model, "saved_model.pt")
```

model = `torch.load(PATH)`

When we load, we load both the network and the model. There is no need for us to create the model first: `new_model2 = Net()` .

In [13]:
```
new_model = torch.load("saved_model.pt")
print(new_model)
```

```
Net(
  (conv1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=14400, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)
```

Caution:

- If you are doing inference, 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.

1.3 Saving the Model Parameters and Optimizer State

It is common to train your model in multiple session where you stop the training temporarily and resume it only at a later day. To do this you need to save **checkpoints**.

When saving a 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:

1. optimizer's state_dict
2. model's state_dict
3. current epoch number
4. training loss
5. others

Assume the following as the current state of training.

In [14]:
```
epoch = 0
model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
loss = np.inf
```

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

In [15]:
```
checkpoint_path = "models/model_epoch" + str(epoch) + ".tar"
```

In [16]:
```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss
            }, checkpoint_path)
```

First, load the *network's parameters* and *optimizer's state*. For the *optimizer*, the learning rate (`lr`) is a compulsory argument. It will be overwritten when we load the saved optimizer's state.

In [17]:
```
new_model = Net()
new_optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Since you wish to resuming training, remember to call `model.train()` to ensure that that the dropout and batch normalization layers are in training mode.

In [18]:
```
checkpoint = torch.load(checkpoint_path)
new_model.load_state_dict(checkpoint['model_state_dict'])
new_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

model.train()  #
```

Out[18]:
```
Net(
  (conv1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=14400, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)
```

Now you are ready to resume your training.

---

2. Example

Load the dataset

We will use the CIFAR10 dataset for example

In [19]:
```
# transform the model
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# dataset
trainset = CIFAR10(train=True,  transform=transform, num_samples=10000)
validset  = CIFAR10(train=False,  transform=transform, num_samples=2000)

# dataloader]
trainloader = DataLoader(trainset, batch_size=32, shuffle=True, num_workers=2)
validloader  = DataLoader(validset, batch_size=128, shuffle=True, num_workers=2)
```

Define training function

First, we define our training model. To allow the model to resume training, we do the following:

1. Define the `model` and `optimizer` outside the `train` function
2. Save the model at the end of each epoch (`line 56` to `62`)

In [20]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

In [21]:
```
def train(model, optimizer, start_epoch=0, max_epochs=10):

    # compute loss 3 times in each epoch
    loss_iterations = int(np.ceil(len(trainloader)/3))

    # transfer model to GPU
    model = model.to(device)

    # set the optimizer. Use SGD with momentum

    # set to training mode
    model.train()

    # train the network
    for e in range(start_epoch, max_epochs):

        running_loss = 0
        running_count = 0

        for i, (inputs, labels) in enumerate(trainloader):

            # Clear all the gradient to 0
            optimizer.zero_grad()

            # transfer data to GPU
            inputs = inputs.to(device)
            labels = labels.to(device)

            # forward propagation to get h
            outs = model(inputs)

            # compute loss
            loss = F.cross_entropy(outs, labels)

            # backpropagation to get gradients of all parameters
            loss.backward()

            # update parameters
            optimizer.step()

            # get the loss
            running_loss += loss.item()
            running_count += 1

             # display the averaged loss value
            if i % loss_iterations == loss_iterations-1 or i == len(trainloader) - 1:
                # compute training loss
                train_loss = running_loss / running_count
                running_loss = 0.
                running_count = 0.

                print(f'[Epoch {e+1:2d} Iter {i+1:5d}/{len(trainloader)}]: train_loss = {train_loss:.4f}')
```

```
        # save the model
        checkpoint_file = './models/saved_model.pt'
        torch.save({
            'epoch': e,
            'loss': train_loss,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            }, checkpoint_file)
```

Train model

Train the model for 2 epochs

In [22]:
```
lr=0.01; momentum=0.9

model = Net()
optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum)

train(model, optimizer, max_epochs=2)
```

```
[Epoch  1 Iter   105/313]: train_loss = 2.0292
[Epoch  1 Iter   210/313]: train_loss = 1.6546
[Epoch  1 Iter   313/313]: train_loss = 1.5681
[Epoch  2 Iter   105/313]: train_loss = 1.3804
[Epoch  2 Iter   210/313]: train_loss = 1.3635
[Epoch  2 Iter   313/313]: train_loss = 1.3252
```

Resume training

Resume training and train for another 2 epochs. To do that, we get the load the *previous* model's and optimizer's `state_dict` , the last epoch and training loss value.

In [23]:
```
# define a new model
new_model = Net()

# define a new optimizer
new_optimizer = optim.SGD(new_model.parameters(), lr=0.1)

# load the checkpoint file
checkpoint = torch.load('./models/saved_model.pt')
new_model.load_state_dict(checkpoint['model_state_dict'])
new_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
previous_epoch = checkpoint['epoch']
previous_loss = checkpoint['loss']

# resume training
print(f'Resuming previous epoch. Last run epoch: {previous_epoch+1}, last run loss: {previous_loss:.4f}')
train(new_model, new_optimizer, start_epoch=previous_epoch+1, max_epochs=4)
```

```
Resuming previous epoch. Last run epoch: 2, last run loss: 1.3252
[Epoch  3 Iter   105/313]: train_loss = 1.1209
[Epoch  3 Iter   210/313]: train_loss = 1.1996
[Epoch  3 Iter   313/313]: train_loss = 1.1479
[Epoch  4 Iter   105/313]: train_loss = 0.8928
[Epoch  4 Iter   210/313]: train_loss = 0.9637
[Epoch  4 Iter   313/313]: train_loss = 0.9826
```

-- END OF LAB ---