Saving and Loading Models

In this notebook, I'll show you how to save and load models with PyTorch. This is important because you'll often want to load previously trained models to use in making predictions or to continue training on new data.

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt

import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
from torchvision import datasets, transforms

import helper
import fc_model
```

In [2]:
```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])
# Download and load the training data
trainset = datasets.FashionMNIST('F_MNIST_data/', download=True, train=True, t
ransform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=Tru
e)

# Download and load the test data
testset = datasets.FashionMNIST('F_MNIST_data/', download=True, train=False, t
ransform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-i
mages-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-l
abels-idx1-ubyte.gz
Processing...
Done!
```

Here we can see one of the images.

In [3]:
```
image, label = next(iter(trainloader))
helper.imshow(image[0,:]);
```

Train a network

To make things more concise here, I moved the model architecture and training code from the last part to a file called `fc_model`. Importing this, we can easily create a fully-connected network with `fc_model.Network`, and train the network using `fc_model.train`. I'll use this model (once it's trained) to demonstrate how we can save and load models.

In [4]:
```
# Create the network, define the criterion and optimizer

model = fc_model.Network(784, 10, [512, 256, 128])
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [5]:
```
fc_model.train(model, trainloader, testloader, criterion, optimizer, epochs=2)
```

```
Epoch: 1/2..  Training Loss: 1.743..  Test Loss: 0.937..  Test Accuracy: 0.66
5
Epoch: 1/2..  Training Loss: 1.039..  Test Loss: 0.738..  Test Accuracy: 0.71
3
Epoch: 1/2..  Training Loss: 0.864..  Test Loss: 0.682..  Test Accuracy: 0.73
0
Epoch: 1/2..  Training Loss: 0.804..  Test Loss: 0.644..  Test Accuracy: 0.75
4
Epoch: 1/2..  Training Loss: 0.775..  Test Loss: 0.602..  Test Accuracy: 0.76
9
Epoch: 1/2..  Training Loss: 0.710..  Test Loss: 0.588..  Test Accuracy: 0.78
3
Epoch: 1/2..  Training Loss: 0.700..  Test Loss: 0.568..  Test Accuracy: 0.78
4
Epoch: 1/2..  Training Loss: 0.659..  Test Loss: 0.568..  Test Accuracy: 0.79
4
Epoch: 1/2..  Training Loss: 0.647..  Test Loss: 0.573..  Test Accuracy: 0.78
2
Epoch: 1/2..  Training Loss: 0.638..  Test Loss: 0.543..  Test Accuracy: 0.80
0
Epoch: 1/2..  Training Loss: 0.641..  Test Loss: 0.541..  Test Accuracy: 0.79
9
Epoch: 1/2..  Training Loss: 0.632..  Test Loss: 0.541..  Test Accuracy: 0.79
7
Epoch: 1/2..  Training Loss: 0.582..  Test Loss: 0.523..  Test Accuracy: 0.80
4
Epoch: 1/2..  Training Loss: 0.611..  Test Loss: 0.528..  Test Accuracy: 0.80
6
Epoch: 1/2..  Training Loss: 0.593..  Test Loss: 0.554..  Test Accuracy: 0.79
9
Epoch: 1/2..  Training Loss: 0.618..  Test Loss: 0.503..  Test Accuracy: 0.81
3
Epoch: 1/2..  Training Loss: 0.582..  Test Loss: 0.511..  Test Accuracy: 0.81
3
Epoch: 1/2..  Training Loss: 0.610..  Test Loss: 0.506..  Test Accuracy: 0.81
5
Epoch: 1/2..  Training Loss: 0.596..  Test Loss: 0.493..  Test Accuracy: 0.81
6
Epoch: 1/2..  Training Loss: 0.530..  Test Loss: 0.495..  Test Accuracy: 0.81
9
Epoch: 1/2..  Training Loss: 0.599..  Test Loss: 0.503..  Test Accuracy: 0.81
5
Epoch: 1/2..  Training Loss: 0.578..  Test Loss: 0.479..  Test Accuracy: 0.82
2
Epoch: 1/2..  Training Loss: 0.532..  Test Loss: 0.486..  Test Accuracy: 0.82
6
Epoch: 2/2..  Training Loss: 0.556..  Test Loss: 0.468..  Test Accuracy: 0.83
0
Epoch: 2/2..  Training Loss: 0.621..  Test Loss: 0.483..  Test Accuracy: 0.83
0
Epoch: 2/2..  Training Loss: 0.536..  Test Loss: 0.472..  Test Accuracy: 0.82
5
Epoch: 2/2..  Training Loss: 0.561..  Test Loss: 0.469..  Test Accuracy: 0.83
3
Epoch: 2/2..  Training Loss: 0.544..  Test Loss: 0.460..  Test Accuracy: 0.82
9
Epoch: 2/2..  Training Loss: 0.519..  Test Loss: 0.473..  Test Accuracy: 0.82
```

```
7
Epoch: 2/2..  Training Loss: 0.546..  Test Loss: 0.458..  Test Accuracy: 0.83
2
Epoch: 2/2..  Training Loss: 0.552..  Test Loss: 0.462..  Test Accuracy: 0.83
4
Epoch: 2/2..  Training Loss: 0.587..  Test Loss: 0.461..  Test Accuracy: 0.83
6
Epoch: 2/2..  Training Loss: 0.518..  Test Loss: 0.460..  Test Accuracy: 0.83
0
Epoch: 2/2..  Training Loss: 0.557..  Test Loss: 0.460..  Test Accuracy: 0.83
5
Epoch: 2/2..  Training Loss: 0.523..  Test Loss: 0.465..  Test Accuracy: 0.82
6
Epoch: 2/2..  Training Loss: 0.508..  Test Loss: 0.464..  Test Accuracy: 0.82
9
Epoch: 2/2..  Training Loss: 0.521..  Test Loss: 0.454..  Test Accuracy: 0.83
7
Epoch: 2/2..  Training Loss: 0.488..  Test Loss: 0.448..  Test Accuracy: 0.84
2
Epoch: 2/2..  Training Loss: 0.510..  Test Loss: 0.448..  Test Accuracy: 0.83
8
Epoch: 2/2..  Training Loss: 0.514..  Test Loss: 0.463..  Test Accuracy: 0.82
5
Epoch: 2/2..  Training Loss: 0.516..  Test Loss: 0.438..  Test Accuracy: 0.84
1
Epoch: 2/2..  Training Loss: 0.516..  Test Loss: 0.444..  Test Accuracy: 0.84
1
Epoch: 2/2..  Training Loss: 0.528..  Test Loss: 0.440..  Test Accuracy: 0.83
9
Epoch: 2/2..  Training Loss: 0.524..  Test Loss: 0.447..  Test Accuracy: 0.83
2
Epoch: 2/2..  Training Loss: 0.502..  Test Loss: 0.430..  Test Accuracy: 0.83
9
Epoch: 2/2..  Training Loss: 0.511..  Test Loss: 0.455..  Test Accuracy: 0.83
1
```

Saving and loading networks

As you can imagine, it's impractical to train a network every time you need to use it. Instead, we can save trained networks then load them later to train more or use them for predictions.

The parameters for PyTorch networks are stored in a model's `state_dict`. We can see the state dict contains the weight and bias matrices for each of our layers.

In [6]:
```
print("Our model: \n\n", model, '\n')
print("The state dict keys: \n\n", model.state_dict().keys())
```

```
Our model:

 Network(
  (hidden_layers): ModuleList(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): Linear(in_features=512, out_features=256, bias=True)
    (2): Linear(in_features=256, out_features=128, bias=True)
  )
  (output): Linear(in_features=128, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)

The state dict keys:

 odict_keys(['hidden_layers.0.weight', 'hidden_layers.0.bias', 'hidden_layer
s.1.weight', 'hidden_layers.1.bias', 'hidden_layers.2.weight', 'hidden_layer
s.2.bias', 'output.weight', 'output.bias'])
```

The simplest thing to do is simply save the state dict with `torch.save`. For example, we can save it to a file `'checkpoint.pth'`.

In [7]:
```
torch.save(model.state_dict(), 'checkpoint.pth')
```

Then we can load the state dict with `torch.load`.

In [8]:
```
state_dict = torch.load('checkpoint.pth')
print(state_dict.keys())
```

```
odict_keys(['hidden_layers.0.weight', 'hidden_layers.0.bias', 'hidden_layers.
1.weight', 'hidden_layers.1.bias', 'hidden_layers.2.weight', 'hidden_layers.
2.bias', 'output.weight', 'output.bias'])
```

And to load the state dict in to the network, you do `model.load_state_dict(state_dict)`.

In [9]:
```
model.load_state_dict(state_dict)
```

Seems pretty straightforward, but as usual it's a bit more complicated. Loading the state dict works only if the model architecture is exactly the same as the checkpoint architecture. If I create a model with a different architecture, this fails.

In [10]:
```
# Try this
model = fc_model.Network(784, 10, [400, 200, 100])
# This will throw an error because the tensor sizes are wrong!
model.load_state_dict(state_dict)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-10-d859c59ebec0> in <module>()
      2 model = fc_model.Network(784, 10, [400, 200, 100])
      3 # This will throw an error because the tensor sizes are wrong!
----> 4 model.load_state_dict(state_dict)

/opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in load_sta
te_dict(self, state_dict, strict)
    719         if len(error_msgs) > 0:
    720             raise RuntimeError('Error(s) in loading state_dict for
 {}:\n\t{}'.format(
--> 721                                self.__class__.__name__, "\n\t".join(e
rror_msgs)))
    722
    723     def parameters(self):

RuntimeError: Error(s) in loading state_dict for Network:
        While copying the parameter named "hidden_layers.0.weight", whose dim
ensions in the model are torch.Size([400, 784]) and whose dimensions in the c
heckpoint are torch.Size([512, 784]).
        While copying the parameter named "hidden_layers.0.bias", whose dimen
sions in the model are torch.Size([400]) and whose dimensions in the checkpoi
nt are torch.Size([512]).
        While copying the parameter named "hidden_layers.1.weight", whose dim
ensions in the model are torch.Size([200, 400]) and whose dimensions in the c
heckpoint are torch.Size([256, 512]).
        While copying the parameter named "hidden_layers.1.bias", whose dimen
sions in the model are torch.Size([200]) and whose dimensions in the checkpoi
nt are torch.Size([256]).
        While copying the parameter named "hidden_layers.2.weight", whose dim
ensions in the model are torch.Size([100, 200]) and whose dimensions in the c
heckpoint are torch.Size([128, 256]).
        While copying the parameter named "hidden_layers.2.bias", whose dimen
sions in the model are torch.Size([100]) and whose dimensions in the checkpoi
nt are torch.Size([128]).
        While copying the parameter named "output.weight", whose dimensions i
n the model are torch.Size([10, 100]) and whose dimensions in the checkpoint
 are torch.Size([10, 128]).
```

This means we need to rebuild the model exactly as it was when trained. Information about the model architecture needs to be saved in the checkpoint, along with the state dict. To do this, you build a dictionary with all the information you need to compeletely rebuild the model.

In [11]:
```
checkpoint = {'input_size': 784,
              'output_size': 10,
              'hidden_layers': [each.out_features for each in model.hidden_lay
ers],
              'state_dict': model.state_dict()}

torch.save(checkpoint, 'checkpoint.pth')
```

Now the checkpoint has all the necessary information to rebuild the trained model. You can easily make that a function if you want. Similarly, we can write a function to load checkpoints.

In [12]:
```
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = fc_model.Network(checkpoint['input_size'],
                             checkpoint['output_size'],
                             checkpoint['hidden_layers'])
    model.load_state_dict(checkpoint['state_dict'])

    return model
```

In [13]:
```
model = load_checkpoint('checkpoint.pth')
print(model)
```

```
Network(
  (hidden_layers): ModuleList(
    (0): Linear(in_features=784, out_features=400, bias=True)
    (1): Linear(in_features=400, out_features=200, bias=True)
    (2): Linear(in_features=200, out_features=100, bias=True)
  )
  (output): Linear(in_features=100, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)
```

In [ ]:
```

```