Recap: What we have been doing

`1-Tensors-in-PyTorch.ipynb`: introduces a tensor representation in PyTorch

`2-Neural-networks-in-PyTorch.ipynb`: introduces a basic framework for defining neural networks via the `nn` module

`3-Training-neural-networks.ipynb`: introduces loss and backprop to improve predictions

`4-Fashion-MNIST.ipynb`: walks through the feed-forward neural network on Fashion MNIST data

`5-Inference-and-Validation.ipynb`: introduces dropout

Saving and loading a model

In this notebook we will learn how to save and load our models. This is handy because we will often want to load previously trained models to use in making predictions or to continue training on new data.

In [25]:
```
import torch
from torchvision import datasets, transforms
from torch import nn, optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import helper
import fc_model
```

In [26]:
```
transform = transforms.Compose([transforms.ToTensor()])

# Download and load the training data
trainset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/', download=True, train=
True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

# Download and load the test data
testset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/', download=True, train=F
alse, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

In [27]:
```
# Here we can see one of the images:
image, label = next(iter(trainloader))
helper.imshow(image[10,:])
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12096fac8>
```

In [28]:
```
label
```

Out[28]:
```
tensor([5, 2, 3, 7, 7, 4, 3, 6, 4, 2, 3, 2, 7, 8, 4, 9, 7, 7, 5, 7, 8, 9, 3, 9,
        4, 9, 1, 6, 6, 6, 9, 6, 3, 1, 4, 9, 3, 3, 0, 2, 8, 6, 3, 0, 2, 7, 3, 8,
        6, 1, 0, 9, 4, 3, 7, 8, 6, 2, 3, 7, 9, 6, 0, 5])
```

In [33]:
```
# Create the network, define the criterion and optimizer

model = fc_model.Network(784, 10, [512, 256, 128, 64])
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [34]:
```
model
```

Out[34]:
```
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)
    (3): Linear(in_features=128, out_features=64, bias=True)
  )
  (output): Linear(in_features=64, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)
```

In [35]:
```
fc_model.train(model, trainloader, testloader, criterion, optimizer, epochs=2)
```

```
Epoch: 1/2..  Training Loss: 2.242..  Test Loss: 1.934..  Test Accuracy: 0.267
Epoch: 1/2..  Training Loss: 1.706..  Test Loss: 1.180..  Test Accuracy: 0.510
Epoch: 1/2..  Training Loss: 1.309..  Test Loss: 0.959..  Test Accuracy: 0.567
Epoch: 1/2..  Training Loss: 1.120..  Test Loss: 0.890..  Test Accuracy: 0.596
Epoch: 1/2..  Training Loss: 1.018..  Test Loss: 0.808..  Test Accuracy: 0.672
Epoch: 1/2..  Training Loss: 0.972..  Test Loss: 0.768..  Test Accuracy: 0.681
Epoch: 1/2..  Training Loss: 0.909..  Test Loss: 0.702..  Test Accuracy: 0.742
Epoch: 1/2..  Training Loss: 0.828..  Test Loss: 0.683..  Test Accuracy: 0.718
Epoch: 1/2..  Training Loss: 0.784..  Test Loss: 0.659..  Test Accuracy: 0.769
Epoch: 1/2..  Training Loss: 0.784..  Test Loss: 0.660..  Test Accuracy: 0.728
Epoch: 1/2..  Training Loss: 0.759..  Test Loss: 0.618..  Test Accuracy: 0.768
Epoch: 1/2..  Training Loss: 0.736..  Test Loss: 0.608..  Test Accuracy: 0.767
Epoch: 1/2..  Training Loss: 0.742..  Test Loss: 0.615..  Test Accuracy: 0.773
Epoch: 1/2..  Training Loss: 0.689..  Test Loss: 0.612..  Test Accuracy: 0.742
Epoch: 1/2..  Training Loss: 0.710..  Test Loss: 0.610..  Test Accuracy: 0.743
Epoch: 1/2..  Training Loss: 0.710..  Test Loss: 0.594..  Test Accuracy: 0.754
Epoch: 1/2..  Training Loss: 0.732..  Test Loss: 0.573..  Test Accuracy: 0.783
Epoch: 1/2..  Training Loss: 0.720..  Test Loss: 0.616..  Test Accuracy: 0.757
Epoch: 1/2..  Training Loss: 0.680..  Test Loss: 0.570..  Test Accuracy: 0.784
Epoch: 1/2..  Training Loss: 0.654..  Test Loss: 0.549..  Test Accuracy: 0.770
Epoch: 1/2..  Training Loss: 0.642..  Test Loss: 0.553..  Test Accuracy: 0.777
Epoch: 1/2..  Training Loss: 0.643..  Test Loss: 0.538..  Test Accuracy: 0.791
Epoch: 1/2..  Training Loss: 0.639..  Test Loss: 0.549..  Test Accuracy: 0.775
Epoch: 2/2..  Training Loss: 0.680..  Test Loss: 0.545..  Test Accuracy: 0.797
Epoch: 2/2..  Training Loss: 0.602..  Test Loss: 0.532..  Test Accuracy: 0.791
Epoch: 2/2..  Training Loss: 0.669..  Test Loss: 0.535..  Test Accuracy: 0.789
Epoch: 2/2..  Training Loss: 0.634..  Test Loss: 0.529..  Test Accuracy: 0.791
Epoch: 2/2..  Training Loss: 0.607..  Test Loss: 0.551..  Test Accuracy: 0.787
Epoch: 2/2..  Training Loss: 0.606..  Test Loss: 0.517..  Test Accuracy: 0.799
Epoch: 2/2..  Training Loss: 0.596..  Test Loss: 0.530..  Test Accuracy: 0.785
Epoch: 2/2..  Training Loss: 0.617..  Test Loss: 0.513..  Test Accuracy: 0.804
Epoch: 2/2..  Training Loss: 0.583..  Test Loss: 0.509..  Test Accuracy: 0.798
Epoch: 2/2..  Training Loss: 0.614..  Test Loss: 0.533..  Test Accuracy: 0.796
Epoch: 2/2..  Training Loss: 0.601..  Test Loss: 0.531..  Test Accuracy: 0.789
Epoch: 2/2..  Training Loss: 0.620..  Test Loss: 0.517..  Test Accuracy: 0.804
Epoch: 2/2..  Training Loss: 0.605..  Test Loss: 0.546..  Test Accuracy: 0.789
Epoch: 2/2..  Training Loss: 0.611..  Test Loss: 0.518..  Test Accuracy: 0.799
Epoch: 2/2..  Training Loss: 0.599..  Test Loss: 0.508..  Test Accuracy: 0.801
Epoch: 2/2..  Training Loss: 0.591..  Test Loss: 0.513..  Test Accuracy: 0.800
Epoch: 2/2..  Training Loss: 0.582..  Test Loss: 0.515..  Test Accuracy: 0.797
Epoch: 2/2..  Training Loss: 0.599..  Test Loss: 0.515..  Test Accuracy: 0.800
Epoch: 2/2..  Training Loss: 0.586..  Test Loss: 0.494..  Test Accuracy: 0.806
Epoch: 2/2..  Training Loss: 0.587..  Test Loss: 0.484..  Test Accuracy: 0.812
Epoch: 2/2..  Training Loss: 0.583..  Test Loss: 0.485..  Test Accuracy: 0.811
Epoch: 2/2..  Training Loss: 0.539..  Test Loss: 0.496..  Test Accuracy: 0.806
Epoch: 2/2..  Training Loss: 0.601..  Test Loss: 0.491..  Test Accuracy: 0.810
```

As we can imagine, it is very impractical to train the network everytime we use it. Instead, we can save trained networks and load them everytime to train them more or to make predictions. The parameters for a PyTorch network are stored in a model's `state.dict`, which contains the weight and bias matrices for each of our layers.

In [39]:
```
model
```

Out[39]:
```
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)
    (3): Linear(in_features=128, out_features=64, bias=True)
  )
  (output): Linear(in_features=64, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)
```

In [37]:
```
model.state_dict()
```

```
Out[37]: OrderedDict([('hidden_layers.0.weight',
                       tensor([[ 0.0354,  0.0118, -0.0117,  ...,  0.0083,  0.0075, -0.014
         0],
                               [ 0.0281, -0.0094,  0.0686,  ..., -0.0598, -0.0028,  0.039
         4],
                               [ 0.0203,  0.0137,  0.0013,  ..., -0.0356,  0.0322,  0.014
         1],
                               ...,
                               [ 0.0583,  0.0457, -0.0042,  ..., -0.0084, -0.0323,  0.003
         8],
                               [ 0.0126, -0.0202,  0.0190,  ...,  0.0055,  0.0213,  0.017
         9],
                               [ 0.0149,  0.0684,  0.0478,  ...,  0.0060, -0.0020,  0.083
         0]])),
                      ('hidden_layers.0.bias',
                       tensor([ 0.0142,  0.0922, -0.0376,  0.0231,  0.0065, -0.0263, -0.0
         205,  0.0459,
                                0.0135,  0.1402, -0.0228, -0.0213, -0.0380, -0.0315, -0.0
         299,  0.1137,
                                0.0154, -0.0350, -0.0743,  0.0621,  0.0689, -0.1075, -0.0
         286, -0.0140,
                               -0.0057, -0.0666, -0.0044, -0.0327,  0.1079,  0.0025, -0.0
         272, -0.0277,
                                0.0239, -0.0063, -0.0306,  0.0270,  0.2355, -0.0215,  0.0
         025,  0.2175,
                                0.0992,  0.0041, -0.0435,  0.0360,  0.0250, -0.0278,  0.0
         039,  0.0172,
                               -0.0105,  0.1399,  0.0044,  0.0821,  0.0321,  0.0175,  0.1
         093,  0.0040,
                                0.2420,  0.2639, -0.0358, -0.0231, -0.0292,  0.0133,  0.1
         151,  0.0072,
                                0.0682,  0.0801,  0.0201, -0.0460, -0.0273,  0.0094,  0.0
         882,  0.0150,
                                0.0026, -0.0333,  0.0882,  0.2056, -0.0082, -0.0224, -0.0
         230,  0.0076,
                                0.0096,  0.0134, -0.0108,  0.0303,  0.0407, -0.0194, -0.0
         141, -0.0108,
                               -0.0110,  0.0394, -0.0160, -0.1097, -0.0190,  0.0555,  0.1
         369, -0.0194,
                               -0.0315,  0.0152, -0.0343, -0.0362,  0.0234,  0.0488,  0.0
         740,  0.1851,
                               -0.0372,  0.0028, -0.0250, -0.0261, -0.0347,  0.0609, -0.0
         285,  0.1200,
                                0.0753,  0.1332,  0.1235, -0.0280, -0.0101,  0.2895, -0.0
         064, -0.0278,
                               -0.0515, -0.0149,  0.0069,  0.0099,  0.0941, -0.0180, -0.0
         362, -0.0239,
                               -0.0773, -0.0289, -0.1049,  0.0142,  0.0108,  0.0540,  0.0
         866, -0.0512,
                                0.0532,  0.0952,  0.1014, -0.0283,  0.1073, -0.0438, -0.0
         023,  0.0728,
                               -0.0043,  0.0197, -0.1930, -0.1115,  0.1307,  0.0340,  0.0
         503,  0.0742,
                                0.0987, -0.0368, -0.1408,  0.0266,  0.0683, -0.0039, -0.0
         310,  0.1028,
                               -0.0414, -0.0003,  0.2366,  0.0022,  0.0677,  0.0011,  0.0
         934, -0.0199,
                               -0.0337, -0.0985,  0.0185,  0.0886, -0.0332,  0.0049, -0.2
         117, -0.0214,
                                0.0625, -0.0523,  0.0209, -0.0592,  0.0583,  0.0098, -0.0
         134, -0.0798,
                               -0.0357, -0.0044,  0.0455,  0.0263, -0.0341,  0.2168,  0.0
         006, -0.0330,
                               -0.0598, -0.1036, -0.0215, -0.0111, -0.0057, -0.0482, -0.0
         029, -0.1085,
```

In [38]:
```
model.state_dict().keys()
```

Out[38]:
```
odict_keys(['hidden_layers.0.weight', 'hidden_layers.0.bias', 'hidden_layers.1.w
eight', 'hidden_layers.1.bias', 'hidden_layers.2.weight', 'hidden_layers.2.bias'
, 'hidden_layers.3.weight', 'hidden_layers.3.bias', 'output.weight', 'output.bia
s'])
```

So the terms containing weight and bias parameters are stored in `model.state_dict()` and this is what we want to save. To save them, we use `torch.save`. For example, we can save them to a file named `checkpoint.pth`.

In [41]:
```
torch.save(model.state_dict(), 'checkpoint.pth')
```

Then we can load the `state_dict()` with `torch.load` method:

In [42]:
```
state_dict_2 = torch.load('checkpoint.pth')
```

In [43]:
```
# just to confirm they are the same weights and biases:
state_dict_2
```

```
Out[43]: OrderedDict([('hidden_layers.0.weight',
                       tensor([[ 0.0354,  0.0118, -0.0117,  ...,  0.0083,  0.0075, -0.014
         0],
                               [ 0.0281, -0.0094,  0.0686,  ..., -0.0598, -0.0028,  0.039
         4],
                               [ 0.0203,  0.0137,  0.0013,  ..., -0.0356,  0.0322,  0.014
         1],
                               ...,
                               [ 0.0583,  0.0457, -0.0042,  ..., -0.0084, -0.0323,  0.003
         8],
                               [ 0.0126, -0.0202,  0.0190,  ...,  0.0055,  0.0213,  0.017
         9],
                               [ 0.0149,  0.0684,  0.0478,  ...,  0.0060, -0.0020,  0.083
         0]])),
                      ('hidden_layers.0.bias',
                       tensor([ 0.0142,  0.0922, -0.0376,  0.0231,  0.0065, -0.0263, -0.0
         205,  0.0459,
                                0.0135,  0.1402, -0.0228, -0.0213, -0.0380, -0.0315, -0.0
         299,  0.1137,
                                0.0154, -0.0350, -0.0743,  0.0621,  0.0689, -0.1075, -0.0
         286, -0.0140,
                               -0.0057, -0.0666, -0.0044, -0.0327,  0.1079,  0.0025, -0.0
         272, -0.0277,
                                0.0239, -0.0063, -0.0306,  0.0270,  0.2355, -0.0215,  0.0
         025,  0.2175,
                                0.0992,  0.0041, -0.0435,  0.0360,  0.0250, -0.0278,  0.0
         039,  0.0172,
                               -0.0105,  0.1399,  0.0044,  0.0821,  0.0321,  0.0175,  0.1
         093,  0.0040,
                                0.2420,  0.2639, -0.0358, -0.0231, -0.0292,  0.0133,  0.1
         151,  0.0072,
                                0.0682,  0.0801,  0.0201, -0.0460, -0.0273,  0.0094,  0.0
         882,  0.0150,
                                0.0026, -0.0333,  0.0882,  0.2056, -0.0082, -0.0224, -0.0
         230,  0.0076,
                                0.0096,  0.0134, -0.0108,  0.0303,  0.0407, -0.0194, -0.0
         141, -0.0108,
                               -0.0110,  0.0394, -0.0160, -0.1097, -0.0190,  0.0555,  0.1
         369, -0.0194,
                               -0.0315,  0.0152, -0.0343, -0.0362,  0.0234,  0.0488,  0.0
         740,  0.1851,
                               -0.0372,  0.0028, -0.0250, -0.0261, -0.0347,  0.0609, -0.0
         285,  0.1200,
                                0.0753,  0.1332,  0.1235, -0.0280, -0.0101,  0.2895, -0.0
         064, -0.0278,
                               -0.0515, -0.0149,  0.0069,  0.0099,  0.0941, -0.0180, -0.0
         362, -0.0239,
                               -0.0773, -0.0289, -0.1049,  0.0142,  0.0108,  0.0540,  0.0
         866, -0.0512,
                                0.0532,  0.0952,  0.1014, -0.0283,  0.1073, -0.0438, -0.0
         023,  0.0728,
                               -0.0043,  0.0197, -0.1930, -0.1115,  0.1307,  0.0340,  0.0
         503,  0.0742,
                                0.0987, -0.0368, -0.1408,  0.0266,  0.0683, -0.0039, -0.0
         310,  0.1028,
                               -0.0414, -0.0003,  0.2366,  0.0022,  0.0677,  0.0011,  0.0
         934, -0.0199,
                               -0.0337, -0.0985,  0.0185,  0.0886, -0.0332,  0.0049, -0.2
         117, -0.0214,
                                0.0625, -0.0523,  0.0209, -0.0592,  0.0583,  0.0098, -0.0
         134, -0.0798,
                               -0.0357, -0.0044,  0.0455,  0.0263, -0.0341,  0.2168,  0.0
         006, -0.0330,
                               -0.0598, -0.1036, -0.0215, -0.0111, -0.0057, -0.0482, -0.0
         029, -0.1085,
```

To load these weights and biases into the network we are currently working with, we should declare:

In [44]:
```
model.load_state_dict(state_dict_2)
```

This is very useful if we create a new model with randomly initialized weights and biases. If we pass an existing `state_dict()` then the random parameters are replaced by ones we had trained previously.

Using pre-trained weights and biases seems straightforward but be careful about one thing:

Loading the `state_dict()` works only if the model architecture is exactly the same as the checkpoint architecture.

If we create a different architecture, this will NOT work.

In [45]:
```
model = Network(784, 10, [400, 200, 100, 50])
# 4 hidden layers with 400, 200, 100, and 50 units
model.load_state_dict(state_dict_2)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-45-ad553ab2f03f> in <module>
      1 model = Network(784, 10, [400, 200, 100, 50])
      2 # 4 hidden layers with 400, 200, 100, and 50 units
----> 3 model.load_state_dict(state_dict_2)

~/Desktop/Deep-learning-pytorch/.env/lib/python3.5/site-packages/torch/nn/module
s/module.py in load_state_dict(self, state_dict, strict)
    717         if len(error_msgs) > 0:
    718             raise RuntimeError('Error(s) in loading state_dict for {}:\n
\t{}'.format(
--> 719                                self.__class__.__name__, "\n\t".join(erro
r_msgs)))
    720
    721     def parameters(self):

RuntimeError: Error(s) in loading state_dict for Network:
        size mismatch for hidden_layers.0.weight: copying a param of torch.Size(
[400, 784]) from checkpoint, where the shape is torch.Size([512, 784]) in curren
t model.
        size mismatch for hidden_layers.0.bias: copying a param of torch.Size([4
00]) from checkpoint, where the shape is torch.Size([512]) in current model.
        size mismatch for hidden_layers.1.weight: copying a param of torch.Size(
[200, 400]) from checkpoint, where the shape is torch.Size([256, 512]) in curren
t model.
        size mismatch for hidden_layers.1.bias: copying a param of torch.Size([2
00]) from checkpoint, where the shape is torch.Size([256]) in current model.
        size mismatch for hidden_layers.2.weight: copying a param of torch.Size(
[100, 200]) from checkpoint, where the shape is torch.Size([128, 256]) in curren
t model.
        size mismatch for hidden_layers.2.bias: copying a param of torch.Size([1
00]) from checkpoint, where the shape is torch.Size([128]) in current model.
        size mismatch for hidden_layers.3.weight: copying a param of torch.Size(
[50, 100]) from checkpoint, where the shape is torch.Size([64, 128]) in current
model.
        size mismatch for hidden_layers.3.bias: copying a param of torch.Size([5
0]) from checkpoint, where the shape is torch.Size([64]) in current model.
        size mismatch for output.weight: copying a param of torch.Size([10, 50])
from checkpoint, where the shape is torch.Size([10, 64]) in current model.
```

This means we need to rebuild the model exactly as it was when we trained it. For this purpose, it's helpful to save the information about the model architecture in the checkpoint along with the `state_dict()`. To do this, we build a dictionary with all information we need to rebuild the model completely:

In [46]:
```
checkpoint = {'input_size': 784,
              'output_size': 10,
              'hidden_layers': [each.out_features for each in model.hidden_layers],
              'state_dict': model.state_dict()}
```

In [47]:
```
torch.save(checkpoint, 'checkpoint.pth')
```

In [48]:
```
checkpoint['input_size']
```

Out[48]:
```
784
```

Now the checkpoint has all information to rebuild the trained model. We can easily make that a function. Similarly, we can write a function to load checkpoints.

In [49]:
```
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = Network(checkpoint['input_size'],
                    checkpoint['output_size'],
                    checkpoint['hidden_layers'])
    model.load_state_dict(checkpoint['state_dict'])
    return model
```

In [50]:
```
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)
    (3): Linear(in_features=100, out_features=50, bias=True)
  )
  (output): Linear(in_features=50, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)
```

In [ ]:
```

```