Classifying Fashion-MNIST

Now it's your turn to build and train a neural network. You'll be using the Fashion-MNIST dataset (https://github.com/zalandoresearch/fashion-mnist), a drop-in replacement for the MNIST dataset. MNIST is actually quite trivial with neural networks where you can easily achieve better than 97% accuracy. Fashion-MNIST is a set of 28x28 greyscale images of clothes. It's more complex than MNIST, so it's a better representation of the actual performance of your network, and a better representation of datasets you'll use in the real world.

<img src='assets/fashion-mnist-sprite.png' width=500px>

In this notebook, you'll build your own neural network. For the most part, you could just copy and paste the code from Part 3, but you wouldn't be learning. It's important for you to write the code yourself and get it to work. Feel free to consult the previous notebooks though as you work through this.

First off, let's load the dataset through torchvision.

In [47]:
```
import torch
from torchvision import datasets, transforms
from torch import nn
import torch.nn.functional as F
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import helper
```

In [48]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [49]:
```
# Define a transform but don't use normalization
#transform = transforms.Compose([transforms.ToTensor(),
#                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])

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 [50]:
```
image, label = next(iter(trainloader))
image.mean(dim = 0).shape
```

Out[50]:
```
torch.Size([1, 28, 28])
```

In [51]:
```
# Average image in a batch:
print(image.mean(dim = 0))
print(image.mean(dim = 0).mean())
```

```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0010, 0.0011, 0.0011, 0.0089,
          0.0203, 0.0499, 0.1295, 0.2156, 0.2336, 0.2169, 0.1732, 0.2081,
          0.2403, 0.1719, 0.0967, 0.0241, 0.0199, 0.0066, 0.0039, 0.0042,
          0.0022, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0013, 0.0012, 0.0220, 0.0710,
          0.1602, 0.2359, 0.3194, 0.4336, 0.4651, 0.4835, 0.4504, 0.4647,
          0.4542, 0.3752, 0.3491, 0.2115, 0.1194, 0.0698, 0.0382, 0.0176,
          0.0165, 0.0088, 0.0002, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0002, 0.0046, 0.0328, 0.0828, 0.1277,
          0.2211, 0.2749, 0.3769, 0.4407, 0.4730, 0.4801, 0.4699, 0.4833,
          0.4374, 0.4369, 0.4071, 0.2873, 0.2062, 0.1502, 0.1039, 0.0513,
          0.0227, 0.0045, 0.0005, 0.0000],
         [0.0000, 0.0001, 0.0005, 0.0003, 0.0213, 0.0675, 0.1188, 0.1838,
          0.2692, 0.3214, 0.3981, 0.4406, 0.4448, 0.4635, 0.4735, 0.4751,
          0.4551, 0.4458, 0.4348, 0.3564, 0.2669, 0.1934, 0.1193, 0.0934,
          0.0282, 0.0019, 0.0005, 0.0001],
         [0.0000, 0.0000, 0.0000, 0.0058, 0.0387, 0.1029, 0.1557, 0.2394,
          0.2836, 0.3525, 0.4330, 0.4270, 0.4414, 0.4641, 0.4732, 0.4612,
          0.4417, 0.4385, 0.4419, 0.4123, 0.3298, 0.2564, 0.1724, 0.1231,
          0.0669, 0.0067, 0.0052, 0.0006],
         [0.0000, 0.0067, 0.0091, 0.0195, 0.0566, 0.1393, 0.2250, 0.2719,
          0.3098, 0.3595, 0.4337, 0.4319, 0.4480, 0.4789, 0.5023, 0.4860,
          0.4633, 0.4495, 0.4790, 0.4537, 0.3860, 0.3272, 0.2672, 0.1767,
          0.1244, 0.0396, 0.0163, 0.0021],
         [0.0000, 0.0183, 0.0308, 0.0561, 0.0804, 0.1746, 0.2651, 0.2958,
          0.3322, 0.3591, 0.4373, 0.4415, 0.4728, 0.4772, 0.5159, 0.5040,
          0.5108, 0.5203, 0.5172, 0.4935, 0.4079, 0.3656, 0.3126, 0.2238,
          0.1652, 0.0936, 0.0591, 0.0057],
         [0.0129, 0.0290, 0.0388, 0.0623, 0.1039, 0.1903, 0.2568, 0.3253,
          0.3858, 0.3501, 0.4490, 0.4645, 0.4864, 0.4935, 0.5333, 0.5387,
          0.5664, 0.5972, 0.5474, 0.5170, 0.4740, 0.3999, 0.3402, 0.2652,
          0.1873, 0.1148, 0.0810, 0.0196],
         [0.0242, 0.0254, 0.0385, 0.0651, 0.1119, 0.2201, 0.2846, 0.3331,
          0.3638, 0.3522, 0.4250, 0.4429, 0.4733, 0.5305, 0.5112, 0.5637,
          0.6140, 0.5757, 0.5364, 0.5206, 0.4792, 0.4298, 0.3973, 0.3218,
          0.2241, 0.1678, 0.1018, 0.0319],
         [0.0268, 0.0499, 0.0454, 0.0688, 0.1400, 0.2464, 0.2849, 0.3420,
          0.3828, 0.3320, 0.4100, 0.4634, 0.4771, 0.5616, 0.5369, 0.6115,
          0.6491, 0.5892, 0.5124, 0.5316, 0.5166, 0.4635, 0.4374, 0.3567,
          0.2573, 0.2231, 0.1178, 0.0325],
         [0.0260, 0.0518, 0.0512, 0.0738, 0.1419, 0.2479, 0.2925, 0.3542,
          0.3842, 0.3241, 0.4124, 0.4695, 0.4927, 0.5771, 0.5560, 0.6093,
          0.6186, 0.5811, 0.4961, 0.5191, 0.5449, 0.4825, 0.4669, 0.3906,
          0.2756, 0.2305, 0.1399, 0.0286],
         [0.0217, 0.0440, 0.0385, 0.0700, 0.1506, 0.2592, 0.2961, 0.3638,
          0.3711, 0.3325, 0.4088, 0.4998, 0.5360, 0.5724, 0.5620, 0.6007,
          0.6276, 0.6034, 0.5189, 0.5403, 0.5600, 0.5100, 0.4745, 0.3880,
          0.2507, 0.2333, 0.1749, 0.0314],
         [0.0256, 0.0391, 0.0403, 0.0688, 0.1370, 0.2694, 0.2971, 0.3604,
          0.3784, 0.3662, 0.4485, 0.5227, 0.5376, 0.5806, 0.5839, 0.6034,
          0.6294, 0.6154, 0.5629, 0.5217, 0.5242, 0.4974, 0.4625, 0.3839,
          0.2703, 0.2370, 0.1756, 0.0406],
         [0.0237, 0.0401, 0.0430, 0.0712, 0.1406, 0.2388, 0.2737, 0.3532,
          0.3872, 0.4078, 0.4839, 0.5544, 0.6064, 0.6104, 0.5805, 0.5940,
          0.6403, 0.6252, 0.5801, 0.4928, 0.4750, 0.4637, 0.4339, 0.3734,
          0.2934, 0.2364, 0.1889, 0.0554],
         [0.0224, 0.0520, 0.0631, 0.0894, 0.1600, 0.2426, 0.2713, 0.3305,
          0.3600, 0.4104, 0.5206, 0.6046, 0.6317, 0.6080, 0.5719, 0.5843,
          0.6466, 0.6222, 0.5993, 0.4890, 0.4314, 0.4597, 0.4461, 0.3806,
          0.3055, 0.2516, 0.1975, 0.0431],
         [0.0287, 0.0712, 0.0730, 0.1012, 0.1779, 0.2698, 0.2979, 0.3181,
          0.3623, 0.4757, 0.5680, 0.6233, 0.6404, 0.6150, 0.5827, 0.6044,
          0.6588, 0.6322, 0.5961, 0.5054, 0.4246, 0.4522, 0.4374, 0.3852,
          0.3052, 0.2608, 0.2214, 0.0356],
         [0.0319, 0.0715, 0.0790, 0.1305, 0.2093, 0.2935, 0.3223, 0.3377,
```

In [52]:
```
helper.imshow(image[7,:]);
```

Building and train the network

Here you should define your network. As with MNIST, each image is 28x28 which is a total of 784 pixels, and there are 10 classes. You should include at least one hidden layer. We suggest you use ReLU activations for the layers and to return the logits or log-softmax from the forward pass. It's up to you how many layers you add and the size of those layers.

Now you should create your network and train it. First you'll want to define the criterion (http://pytorch.org/docs/master/nn.html#loss-functions) (something like `nn.CrossEntropyLoss`) and the optimizer (http://pytorch.org/docs/master/optim.html) (typically `optim.SGD` or `optim.Adam`).

Then write the training code. Remember the training pass is a fairly straightforward process:

- Make a forward pass through the network to get the logits
- Use the logits to calculate the loss
- Perform a backward pass through the network with `loss.backward()` to calculate the gradients
- Take a step with the optimizer to update the weights

By adjusting the hyperparameters (hidden units, learning rate, etc), you should be able to get the training loss below 0.4.

In [53]:
```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

model
```

Out[53]:
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax()
)
```

In [54]:
```
class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
        # Define ReLU activation and log-softmax output
        self.ReLU = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each of our operations
        x = self.fc1(x)
        x = self.ReLU(x)
        x = self.fc2(x)
        x = self.ReLU(x)
        x = self.fc3(x)
        x = self.logsoftmax(x)
        return x

model = Network()
model
```

Out[54]:
```
Network(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
  (ReLU): ReLU()
  (logsoftmax): LogSoftmax()
)
```

In [55]:
```
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr = 0.001)

epochs = 10
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad()
        # Forward pass, get our log-probabilities
        log_prob = model.forward(images)
        loss = criterion(log_prob, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    else:
        print("Training loss for epoch # " + str(e) + ": " + str(running_loss/len(
trainloader)))
```

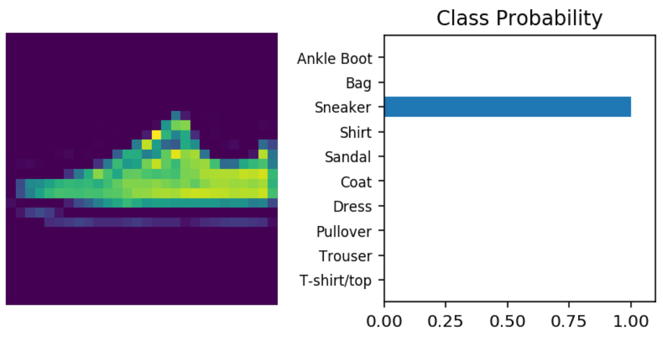
```
Training loss for epoch # 0: 0.5694557493334131
Training loss for epoch # 1: 0.39723556217894374
Training loss for epoch # 2: 0.3556579046014911
Training loss for epoch # 3: 0.32601035233817377
Training loss for epoch # 4: 0.30795767576074295
Training loss for epoch # 5: 0.2953592455749319
Training loss for epoch # 6: 0.2802769504090362
Training loss for epoch # 7: 0.2704023537494099
Training loss for epoch # 8: 0.2577587871877814
Training loss for epoch # 9: 0.2516948783845663
```

In [56]:
```
# Test out your network!
dataiter = iter(testloader)
images, labels = dataiter.next()

# look at a random image
image_index = 3
img = images[image_index]
# Convert 2D image to 1D vector
img = img.resize_(1, 784)

# In the test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    log_probabilities = model.forward(img)
    # Output of the network are logits, need to take softmax for probabilities
    probabilities = torch.exp(log_probabilities)


helper.view_classify(img.resize_(1, 28, 28), probabilities, version='Fashion')
```



In [58]:
```
# Calculating test accuracy:
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in testloader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: {} %'.format(100 * correc
t / total))
```

```
Accuracy of the network on the 10000 test images: 88.05 %
```

In [61]:
```
total
```

Out[61]:
```
10000
```

In [ ]: