Inference and Validation

Now that you have a trained network, you can use it for making predictions. This is typically called **inference**, a term borrowed from statistics. However, neural networks have a tendency to perform *too well* on the training data and aren't able to generalize to data that hasn't been seen before. This is called **overfitting** and it impairs inference performance. To test for overfitting while training, we measure the performance on data not in the training set called the **validation** set. We avoid overfitting through regularization such as dropout while monitoring the validation performance during training. In this notebook, I'll show you how to do this in PyTorch.

As usual, let's start by loading the dataset through torchvision. You'll learn more about torchvision and loading data in a later part. This time we'll be taking advantage of the test set which you can get by setting `train=False` here:

```
testset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/', download=True, train=Fa
lse, transform=transform)
```

The test set contains images just like the training set. Typically you'll see 10-20% of the original dataset held out for testing and validation with the rest being used for training.

In [1]:
```
import torch
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.
5, 0.5))])
# Download and load the training data
trainset = datasets.FashionMNIST('~/.pytorch/F_MNIST_data/', download=True, tr
ain=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=Tru
e)

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

Here I'll create a model like normal, using the same one from my solution for part 4.

In [2]:
```
from torch import nn, optim
import torch.nn.functional as F

class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

The goal of validation is to measure the model's performance on data that isn't part of the training set. Performance here is up to the developer to define though. Typically this is just accuracy, the percentage of classes the network predicted correctly. Other options are [precision and recall](https://en.wikipedia.org/wiki/Precision_and_recall#Definition_(classification_context)) and top-5 error rate. We'll focus on accuracy here. First I'll do a forward pass with one batch from the test set.

In [3]:
```
model = Classifier()

images, labels = next(iter(testloader))
# Get the class probabilities
ps = torch.exp(model(images))
# Make sure the shape is appropriate, we should get 10 class probabilities for
64 examples
print(ps.shape)
```

```
torch.Size([64, 10])
```

With the probabilities, we can get the most likely class using the `ps.topk` method. This returns the k highest values. Since we just want the most likely class, we can use `ps.topk(1)`. This returns a tuple of the top-k values and the top-k indices. If the highest value is the fifth element, we'll get back 4 as the index.

In [4]:
```
top_p, top_class = ps.topk(1, dim=1)
# Look at the most likely classes for the first 10 examples
print(top_class[:10,:])
```

```
tensor([[ 8],
        [ 8],
        [ 7],
        [ 8],
        [ 2],
        [ 8],
        [ 8],
        [ 8],
        [ 8],
        [ 8]])
```

Now we can check if the predicted classes match the labels. This is simple to do by equating `top_class` and `labels`, but we have to be careful of the shapes. Here `top_class` is a 2D tensor with shape `(64, 1)` while `labels` is 1D with shape `(64)`. To get the equality to work out the way we want, `top_class` and `labels` must have the same shape.

If we do

```
equals = top_class == labels
```

`equals` will have shape `(64, 64)`, try it yourself. What it's doing is comparing the one element in each row of `top_class` with each element in `labels` which returns 64 True/False boolean values for each row.

In [5]:
```
equals = top_class == labels.view(*top_class.shape)
```

Now we need to calculate the percentage of correct predictions. `equals` has binary values, either 0 or 1. This means that if we just sum up all the values and divide by the number of values, we get the percentage of correct predictions. This is the same operation as taking the mean, so we can get the accuracy with a call to `torch.mean`. If only it was that simple. If you try `torch.mean(equals)`, you'll get an error

```
RuntimeError: mean is not implemented for type torch.ByteTensor
```

This happens because `equals` has type `torch.ByteTensor` but `torch.mean` isn't implemented for tensors with that type. So we'll need to convert `equals` to a float tensor. Note that when we take `torch.mean` it returns a scalar tensor, to get the actual value as a float we'll need to do `accuracy.item()`.

In [6]:
```
accuracy = torch.mean(equals.type(torch.FloatTensor))
print(f'Accuracy: {accuracy.item()*100}%')
```

```
Accuracy: 7.8125%
```

The network is untrained so it's making random guesses and we should see an accuracy around 10%. Now let's train our network and include our validation pass so we can measure how well the network is performing on the test set. Since we're not updating our parameters in the validation pass, we can speed up our code by turning off gradients using `torch.no_grad()`:

```
# turn off gradients
with torch.no_grad():
    # validation pass here
    for images, labels in testloader:
        ...
```

> **Exercise:** Implement the validation loop below and print out the total accuracy after the loop. You can largely copy and paste the code from above, but I suggest typing it in because writing it out yourself is essential for building the skill. In general you'll always learn more by typing it rather than copy-pasting. You should be able to get an accuracy above 80%.

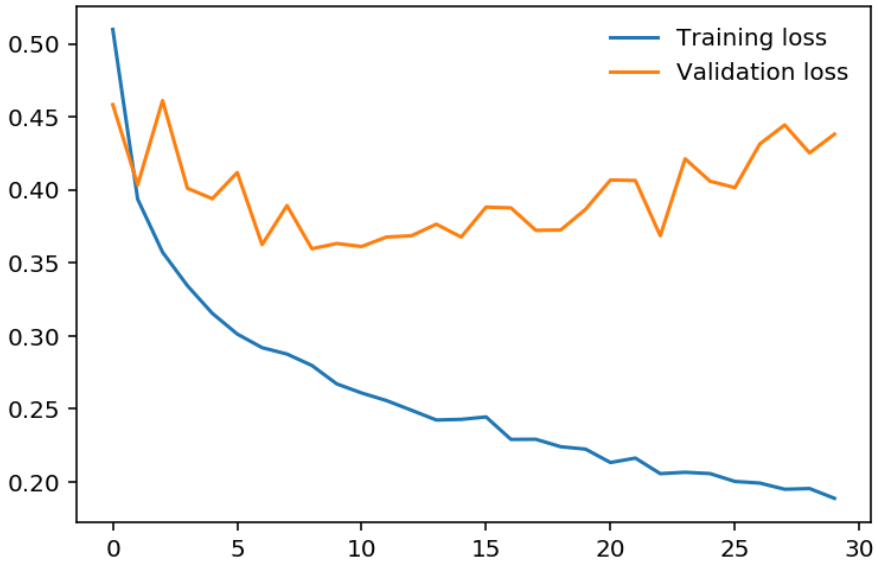
In [7]:

```
model = Classifier()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)

epochs = 8
steps = 0

train_losses, test_losses = [], []
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:

        optimizer.zero_grad()

        log_ps = model(images)
        loss = criterion(log_ps, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    else:
        ## TODO: Implement the validation pass and print out the validation ac
curacy
        accuracy = 0
        test_loss = 0

        with torch.no_grad():
            for images, labels in testloader:
                log_ps = model(images)
                loss = criterion(log_ps, labels)
                ps = torch.exp(log_ps)
                top_p, top_class = ps.topk(1, dim=1)

                equals = top_class == labels.view(*top_class.shape)
                accuracy += torch.mean(equals.type(torch.FloatTensor))
                test_loss += loss.item()

        accuracy /= len(testloader)

        print(f'Epoch {e}: ',
              f'Training loss: {running_loss/len(trainloader)}',
              f'Testing loss: {test_loss/len(testloader)}',
              f'Accuracy: {accuracy.item()*100}%')
```

```
Epoch 0:  Training loss: 0.5136879956258386 Testing loss: 0.44956651149661675
Accuracy: 83.0911636352539%
Epoch 1:  Training loss: 0.3949364596910314 Testing loss: 0.4231641265047584
Accuracy: 84.54418778419495%
Epoch 2:  Training loss: 0.3529848268211905 Testing loss: 0.3850349338760801
Accuracy: 86.49482727050781%
Epoch 3:  Training loss: 0.3343661492948593 Testing loss: 0.38267457039113256
Accuracy: 85.92754602432251%
Epoch 4:  Training loss: 0.31644013898173123 Testing loss: 0.3845800077839262
Accuracy: 86.63415312767029%
Epoch 5:  Training loss: 0.3030921278048807 Testing loss: 0.38841367171258684
Accuracy: 86.6042971611023%
Epoch 6:  Training loss: 0.2924045009383642 Testing loss: 0.39610278440319047
Accuracy: 86.96258068084717%
Epoch 7:  Training loss: 0.28077785135395744 Testing loss: 0.4146121376828783
Accuracy: 86.09673380851746%
```

Overfitting

If we look at the training and validation losses as we train the network, we can see a phenomenon known as overfitting.



The network learns the training set better and better, resulting in lower training losses. However, it starts having problems generalizing to data outside the training set leading to the validation loss increasing. The ultimate goal of any deep learning model is to make predictions on new data, so we should strive to get the lowest validation loss possible. One option is to use the version of the model with the lowest validation loss, here the one around 8-10 training epochs. This strategy is called *early-stopping*. In practice, you'd save the model frequently as you're training then later choose the model with the lowest validation loss.

The most common method to reduce overfitting (outside of early-stopping) is *dropout*, where we randomly drop input units. This forces the network to share information between weights, increasing it's ability to generalize to new data. Adding dropout in PyTorch is straightforward using the `nn.Dropout` (https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout) module.

```
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

        # Dropout module with 0.2 drop probability
        self.dropout = nn.Dropout(p=0.2)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        # Now with dropout
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.dropout(F.relu(self.fc3(x)))

        # output so no dropout here
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

During training we want to use dropout to prevent overfitting, but during inference we want to use the entire network. So, we need to turn off dropout during validation, testing, and whenever we're using the network to make predictions. To do this, you use `model.eval()`. This sets the model to evaluation mode where the dropout probability is 0. You can turn dropout back on by setting the model to train mode with `model.train()`. In general, the pattern for the validation loop will look like this, where you turn off gradients, set the model to

> **Exercise:** Add dropout to your model and train it on Fashion-MNIST again. See if you can get a lower validation loss or higher accuracy.

In [8]:

```
## TODO: Define your model with dropout added
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

        self.dropout = nn.Dropout(p = 0.2)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.dropout(F.relu(self.fc3(x)))
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

In [9]:

```
model = Classifier()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)

epochs = 30
steps = 0

train_losses, test_losses = [], []
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:

        optimizer.zero_grad()

        log_ps = model(images)
        loss = criterion(log_ps, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    else:
        test_loss = 0
        accuracy = 0

        # Turn off gradients for validation, saves memory and computations
        with torch.no_grad():
            model.eval()
            for images, labels in testloader:
                log_ps = model(images)
                test_loss += criterion(log_ps, labels)

                ps = torch.exp(log_ps)
                top_p, top_class = ps.topk(1, dim=1)
                equals = top_class == labels.view(*top_class.shape)
                accuracy += torch.mean(equals.type(torch.FloatTensor))

        model.train()

        train_losses.append(running_loss/len(trainloader))
        test_losses.append(test_loss/len(testloader))

        print("Epoch: {}/{}.. ".format(e+1, epochs),
              "Training Loss: {:.3f}.. ".format(running_loss/len(trainloader
)),
              "Test Loss: {:.3f}.. ".format(test_loss/len(testloader)),
              "Test Accuracy: {:.3f}".format(accuracy/len(testloader)))
```

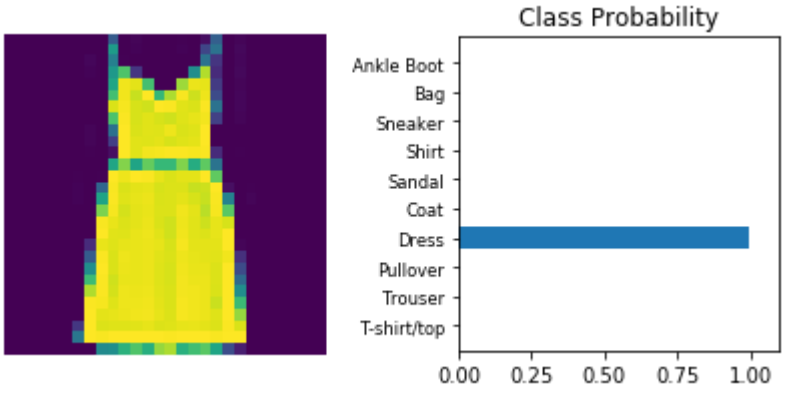
```
Epoch: 1/30..  Training Loss: 0.601..  Test Loss: 0.472..  Test Accuracy: 0.8
23
Epoch: 2/30..  Training Loss: 0.484..  Test Loss: 0.439..  Test Accuracy: 0.8
37
Epoch: 3/30..  Training Loss: 0.450..  Test Loss: 0.449..  Test Accuracy: 0.8
39
Epoch: 4/30..  Training Loss: 0.433..  Test Loss: 0.402..  Test Accuracy: 0.8
55
Epoch: 5/30..  Training Loss: 0.418..  Test Loss: 0.399..  Test Accuracy: 0.8
52
Epoch: 6/30..  Training Loss: 0.409..  Test Loss: 0.415..  Test Accuracy: 0.8
57
Epoch: 7/30..  Training Loss: 0.404..  Test Loss: 0.406..  Test Accuracy: 0.8
57
Epoch: 8/30..  Training Loss: 0.399..  Test Loss: 0.400..  Test Accuracy: 0.8
60
Epoch: 9/30..  Training Loss: 0.389..  Test Loss: 0.393..  Test Accuracy: 0.8
62
Epoch: 10/30..  Training Loss: 0.394..  Test Loss: 0.385..  Test Accuracy: 0.
861
Epoch: 11/30..  Training Loss: 0.382..  Test Loss: 0.387..  Test Accuracy: 0.
858
Epoch: 12/30..  Training Loss: 0.382..  Test Loss: 0.393..  Test Accuracy: 0.
854
Epoch: 13/30..  Training Loss: 0.373..  Test Loss: 0.384..  Test Accuracy: 0.
869
Epoch: 14/30..  Training Loss: 0.377..  Test Loss: 0.385..  Test Accuracy: 0.
870
Epoch: 15/30..  Training Loss: 0.367..  Test Loss: 0.400..  Test Accuracy: 0.
863
Epoch: 16/30..  Training Loss: 0.372..  Test Loss: 0.380..  Test Accuracy: 0.
864
Epoch: 17/30..  Training Loss: 0.364..  Test Loss: 0.371..  Test Accuracy: 0.
874
Epoch: 18/30..  Training Loss: 0.362..  Test Loss: 0.377..  Test Accuracy: 0.
871
Epoch: 19/30..  Training Loss: 0.366..  Test Loss: 0.390..  Test Accuracy: 0.
869
Epoch: 20/30..  Training Loss: 0.357..  Test Loss: 0.388..  Test Accuracy: 0.
867
Epoch: 21/30..  Training Loss: 0.357..  Test Loss: 0.387..  Test Accuracy: 0.
870
Epoch: 22/30..  Training Loss: 0.353..  Test Loss: 0.387..  Test Accuracy: 0.
871
Epoch: 23/30..  Training Loss: 0.358..  Test Loss: 0.385..  Test Accuracy: 0.
868
Epoch: 24/30..  Training Loss: 0.352..  Test Loss: 0.396..  Test Accuracy: 0.
870
Epoch: 25/30..  Training Loss: 0.351..  Test Loss: 0.396..  Test Accuracy: 0.
864
Epoch: 26/30..  Training Loss: 0.347..  Test Loss: 0.397..  Test Accuracy: 0.
869
Epoch: 27/30..  Training Loss: 0.348..  Test Loss: 0.387..  Test Accuracy: 0.
875
Epoch: 28/30..  Training Loss: 0.347..  Test Loss: 0.378..  Test Accuracy: 0.
868
Epoch: 29/30..  Training Loss: 0.340..  Test Loss: 0.368..  Test Accuracy: 0.
```

```
874
Epoch: 30/30..  Training Loss: 0.339..  Test Loss: 0.362..  Test Accuracy: 0.
876
```

Inference

Now that the model is trained, we can use it for inference. We've done this before, but now we need to remember to set the model in inference mode with `model.eval()`. You'll also want to turn off autograd with the `torch.no_grad()` context.

In [11]:
```
# Import helper module (should be in the repo)
import helper

# Test out your network!

model.eval()

dataiter = iter(testloader)
images, labels = dataiter.next()
img = images[0]
# Convert 2D image to 1D vector
img = img.view(1, 784)

# Calculate the class probabilities (softmax) for img
with torch.no_grad():
    output = model.forward(img)

ps = torch.exp(output)

# Plot the image and probabilities
helper.view_classify(img.view(1, 28, 28), ps, version='Fashion')
```



Next Up!

In the next part, I'll show you how to save your trained models. In general, you won't want to train a model everytime you need it. Instead, you'll train once, save it, then load the model when you want to train more or use if for inference.