Classify FashionMNIST, solution 1

May 26, 2020

0.1 # CNN for Classification

In this and the next notebook, we define **and train** a CNN to classify images from the Fashion-MNIST database.

We are providing two solutions to show you how different network structures and training strategies can affect the performance and accuracy of a CNN. This first solution will be a simple CNN with two convolutional layers.

Please note that this is just one possible solution out of many!

0.1.1 Load the data

In this cell, we load in both **training and test** datasets from the FashionMNIST class.

```
In [1]: # our basic libraries
        import torch
        import torchvision

        # data loading and transforming
        from torchvision.datasets import FashionMNIST
        from torch.utils.data import DataLoader
        from torchvision import transforms

        # The output of torchvision datasets are PILImage images of range [0, 1].
        # We transform them to Tensors for input into a CNN

        ## Define a transform to read the data in as a tensor
        data_transform = transforms.ToTensor()

        # choose the training and test datasets
        train_data = FashionMNIST(root='./data', train=True,
                                          download=True, transform=data_transform)

        test_data = FashionMNIST(root='./data', train=False,
                                         download=True, transform=data_transform)


        # Print out some stats about the training and test data
        print('Train data, number of images: ', len(train_data))
        print('Test data, number of images: ', len(test_data))
```

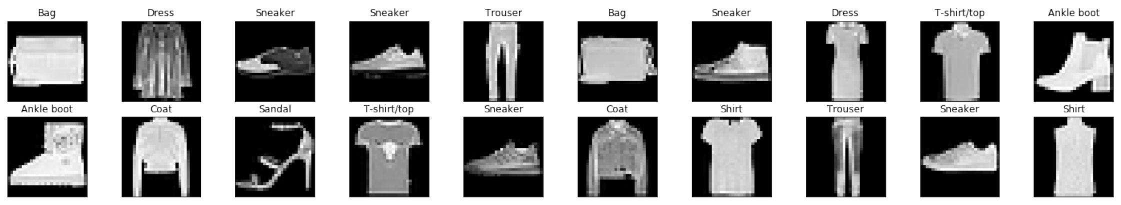
```
Train data, number of images:  60000
Test data, number of images:  10000
```

```
In [2]: # prepare data loaders, set the batch_size
        ## TODO: you can try changing the batch_size to be larger or smaller
        ## when you get to training your network, see how batch_size affects the loss
        batch_size = 20

        train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
        test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=True)

        # specify the image classes
        classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                   'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

0.1.2 Visualize some training data

This cell iterates over the training dataset, loading a random batch of image/label data, using `dataiter.next()`. It then plots the batch of images and labels in a 2 `x` `batch_size/2` grid.

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline

        # obtain one batch of training images
        dataiter = iter(train_loader)
        images, labels = dataiter.next()
        images = images.numpy()

        # plot the images in the batch, along with the corresponding labels
        fig = plt.figure(figsize=(25, 4))
        for idx in np.arange(batch_size):
            ax = fig.add_subplot(2, batch_size/2, idx+1, xticks=[], yticks=[])
            ax.imshow(np.squeeze(images[idx]), cmap='gray')
            ax.set_title(classes[labels[idx]])
```



0.1.3 Define the network architecture

The various layers that make up any neural network are documented, here. For a convolutional neural network, we'll use a simple series of layers: * Convolutional layers * Maxpooling layers * Fully-connected (linear) layers

You are also encouraged to look at adding dropout layers to avoid overfitting this data.

To define a neural network in PyTorch, you define the layers of a model in the function `__init__` and define the feedforward behavior of a network that employs those initialized layers in the function `forward`, which takes in an input image tensor, `x`. The structure of this Net class is shown below and left for you to fill in.

Note: During training, PyTorch will be able to perform backpropagation by keeping track of the network's feedforward behavior and using autograd to calculate the update to the weights in the network.

Define the Layers in `__init__` As a reminder, a conv/pool layer may be defined like this (in `__init__`):

```
# 1 input image channel (for grayscale images), 32 output channels/feature maps, 3x3 square conv
self.conv1 = nn.Conv2d(1, 32, 3)

# maxpool that uses a square window of kernel_size=2, stride=2
self.pool = nn.MaxPool2d(2, 2)
```

Refer to Layers in `forward` Then referred to in the `forward` function like this, in which the conv1 layer has a ReLu activation applied to it before maxpooling is applied:

```
x = self.pool(F.relu(self.conv1(x)))
```

You must place any layers with trainable weights, such as convolutional layers, in the `__init__` function and refer to them in the `forward` function; any layers or functions that always behave in the same way, such as a pre-defined activation function, may appear *only* in the `forward` function. In practice, you'll often see conv/pool layers defined in `__init__` and activations defined in `forward`.

Convolutional layer The first convolution layer has been defined for you, it takes in a 1 channel (grayscale) image and outputs 10 feature maps as output, after convolving the image with 3x3 filters.

Flattening Recall that to move from the output of a convolutional/pooling layer to a linear layer, you must first flatten your extracted features into a vector. If you've used the deep learning library, Keras, you may have seen this done by `Flatten()`, and in PyTorch you can flatten an input `x` with `x = x.view(x.size(0), -1)`.

0.1.4 TODO: Define the rest of the layers

It will be up to you to define the other layers in this network; we have some recommendations, but you may change the architecture and parameters as you see fit.

Recommendations/tips: * Use at least two convolutional layers * Your output must be a linear layer with 10 outputs (for the 10 classes of clothing) * Use a dropout layer to avoid overfitting

0.1.5 A note on output size

For any convolutional layer, the output feature maps will have the specified depth (a depth of 10 for 10 filters in a convolutional layer) and the dimensions of the produced feature maps (width/height) can be computed as the *input image* width/height, W, minus the filter size, F, divided by the stride, S, all + 1. The equation looks like: `output_dim = (W-F)/S + 1`, for an assumed padding size of 0. You can find a derivation of this formula, here.

For a pool layer with a size 2 and stride 2, the output dimension will be reduced by a factor of 2. Read the comments in the code below to see the output size for each layer.

```
In [4]: import torch.nn as nn
        import torch.nn.functional as F

        class Net(nn.Module):

            def __init__(self):
                super(Net, self).__init__()

                # 1 input image channel (grayscale), 10 output channels/feature maps
                # 3x3 square convolution kernel
                ## output size = (W-F)/S +1 = (28-3)/1 +1 = 26
                # the output Tensor for one image, will have the dimensions: (10, 26, 26)
                # after one pool layer, this becomes (10, 13, 13)
                self.conv1 = nn.Conv2d(1, 10, 3)

                # maxpool layer
                # pool with kernel_size=2, stride=2
                self.pool = nn.MaxPool2d(2, 2)

                # second conv layer: 10 inputs, 20 outputs, 3x3 conv
                ## output size = (W-F)/S +1 = (13-3)/1 +1 = 11
                # the output tensor will have dimensions: (20, 11, 11)
                # after another pool layer this becomes (20, 5, 5); 5.5 is rounded down
                self.conv2 = nn.Conv2d(10, 20, 3)

                # 20 outputs * the 5*5 filtered/pooled map size
                # 10 output channels (for the 10 classes)
                self.fc1 = nn.Linear(20*5*5, 10)


            # define the feedforward behavior
            def forward(self, x):
                # two conv/relu + pool layers
                x = self.pool(F.relu(self.conv1(x)))
                x = self.pool(F.relu(self.conv2(x)))

                # prep for linear layer
                # flatten the inputs into a vector
                x = x.view(x.size(0), -1)

                # one linear layer
                x = F.relu(self.fc1(x))
                # a softmax layer to convert the 10 outputs into a distribution of class scores
                x = F.log_softmax(x, dim=1)

                # final output
                return x

        # instantiate and print your Net
        net = Net()
        print(net)
```

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=500, out_features=10, bias=True)
)
```

0.1.6 TODO: Specify the loss function and optimizer

Learn more about loss functions and optimizers in the online documentation.

Note that for a classification problem like this, one typically uses cross entropy loss, which can be defined in code like: `criterion = nn.CrossEntropyLoss()`; cross entropy loss combines `softmax` and `NLL loss` so, alternatively (as in this example), you may see NLL Loss being used when the output of our Net is a distribution of class scores.

PyTorch also includes some standard stochastic optimizers like stochastic gradient descent and Adam. You're encouraged to try different optimizers and see how your model responds to these choices as it trains.

```
In [5]: import torch.optim as optim

        ## TODO: specify loss function
        # cross entropy loss combines softmax and nn.NLLLoss() in one single class.
        criterion = nn.NLLLoss()

        ## TODO: specify optimizer
        # stochastic gradient descent with a small learning rate
        optimizer = optim.SGD(net.parameters(), lr=0.001)
```

0.1.7 A note on accuracy

It's interesting to look at the accuracy of your network **before and after** training. This way you can really see that your network has learned something. In the next cell, let's see what the accuracy of an untrained network is (we expect it to be around 10% which is the same accuracy as just guessing for all 10 classes).

```
In [6]: # Calculate accuracy before training
        correct = 0
        total = 0

        # Iterate through test dataset
        for images, labels in test_loader:

            # forward pass to get outputs
            # the outputs are a series of class scores
            outputs = net(images)

            # get the predicted class from the maximum value in the output-list of class scores
            _, predicted = torch.max(outputs.data, 1)

            # count up total number of correct labels
            # for which the predicted and true labels are equal
            total += labels.size(0)
            correct += (predicted == labels).sum()

        # calculate the accuracy
        # to convert `correct` from a Tensor into a scalar, use .item()
        accuracy = 100.0 * correct.item() / total

        # print it out!
        print('Accuracy before training: ', accuracy)
```

```
Accuracy before training:  9.58
```

0.1.8 Train the Network

Below, we've defined a `train` function that takes in a number of epochs to train for. * The number of epochs is how many times a network will cycle through the entire training dataset. * Inside the epoch loop, we loop over the training dataset in batches; recording the loss every 1000 batches.

Here are the steps that this training function performs as it iterates over the training dataset:

1. Zero's the gradients to prepare for a forward pass
2. Passes the input through the network (forward pass)
3. Computes the loss (how far is the predicted classes are from the correct labels)
4. Propagates gradients back into the network's parameters (backward pass)
5. Updates the weights (parameter update)
6. Prints out the calculated loss

```
In [7]: def train(n_epochs):

            loss_over_time = [] # to track the loss as the network trains

            for epoch in range(n_epochs):  # loop over the dataset multiple times

                running_loss = 0.0

                for batch_i, data in enumerate(train_loader):
                    # get the input images and their corresponding labels
                    inputs, labels = data

                    # zero the parameter (weight) gradients
                    optimizer.zero_grad()

                    # forward pass to get outputs
                    outputs = net(inputs)

                    # calculate the loss
                    loss = criterion(outputs, labels)

                    # backward pass to calculate the parameter gradients
                    loss.backward()

                    # update the parameters
                    optimizer.step()

                    # print loss statistics
                    # to convert loss into a scalar and add it to running_loss, we use .item()
                    running_loss += loss.item()

                    if batch_i % 1000 == 999:    # print every 1000 batches
                        avg_loss = running_loss/1000
                        # record and print the avg loss over the 1000 batches
                        loss_over_time.append(avg_loss)
                        print('Epoch: {}, Batch: {}, Avg. Loss: {}'.format(epoch + 1, batch_i+1,
                        running_loss = 0.0

            print('Finished Training')
            return loss_over_time
```

```
In [8]: # define the number of epochs to train for
        n_epochs = 30 # start small to see if your model works, initially

        # call train and record the loss over time
        training_loss = train(n_epochs)
```

```
Epoch: 1, Batch: 1000, Avg. Loss: 2.2897805507183073
Epoch: 1, Batch: 2000, Avg. Loss: 2.2445739927291872
```

```
Epoch: 1, Batch: 3000, Avg. Loss: 2.1173057613372803
Epoch: 2, Batch: 1000, Avg. Loss: 1.7599379895925522
Epoch: 2, Batch: 2000, Avg. Loss: 1.3709682533144951
Epoch: 2, Batch: 3000, Avg. Loss: 1.1737611926794052
Epoch: 3, Batch: 1000, Avg. Loss: 1.089407693207264
Epoch: 3, Batch: 2000, Avg. Loss: 1.0263702775239945
Epoch: 3, Batch: 3000, Avg. Loss: 0.9977191424965859
Epoch: 4, Batch: 1000, Avg. Loss: 0.9786616771519184
Epoch: 4, Batch: 2000, Avg. Loss: 0.9537102030217648
Epoch: 4, Batch: 3000, Avg. Loss: 0.9409029725492001
Epoch: 5, Batch: 1000, Avg. Loss: 0.9250103669464588
Epoch: 5, Batch: 2000, Avg. Loss: 0.9120527465641498
Epoch: 5, Batch: 3000, Avg. Loss: 0.918829822331667
Epoch: 6, Batch: 1000, Avg. Loss: 0.896810563057661
Epoch: 6, Batch: 2000, Avg. Loss: 0.8878745516836644
Epoch: 6, Batch: 3000, Avg. Loss: 0.8948878626674414
Epoch: 7, Batch: 1000, Avg. Loss: 0.8829183314889669
Epoch: 7, Batch: 2000, Avg. Loss: 0.8684994752109051
Epoch: 7, Batch: 3000, Avg. Loss: 0.8745157381296158
Epoch: 8, Batch: 1000, Avg. Loss: 0.866172359019518
Epoch: 8, Batch: 2000, Avg. Loss: 0.847797974690795
Epoch: 8, Batch: 3000, Avg. Loss: 0.8693051651120186
Epoch: 9, Batch: 1000, Avg. Loss: 0.8467067533284426
Epoch: 9, Batch: 2000, Avg. Loss: 0.854233425244689
Epoch: 9, Batch: 3000, Avg. Loss: 0.8476001562774181
Epoch: 10, Batch: 1000, Avg. Loss: 0.8470250733792782
Epoch: 10, Batch: 2000, Avg. Loss: 0.8427207006663084
Epoch: 10, Batch: 3000, Avg. Loss: 0.8311049311161042
Epoch: 11, Batch: 1000, Avg. Loss: 0.8291002784669399
Epoch: 11, Batch: 2000, Avg. Loss: 0.8266905436888338
Epoch: 11, Batch: 3000, Avg. Loss: 0.8397854830995202
Epoch: 12, Batch: 1000, Avg. Loss: 0.8289510119855404
Epoch: 12, Batch: 2000, Avg. Loss: 0.8189487120509148
Epoch: 12, Batch: 3000, Avg. Loss: 0.8256137864552439
Epoch: 13, Batch: 1000, Avg. Loss: 0.8208291070610285
Epoch: 13, Batch: 2000, Avg. Loss: 0.8178811836466193
Epoch: 13, Batch: 3000, Avg. Loss: 0.8141642866879701
Epoch: 14, Batch: 1000, Avg. Loss: 0.8176477542072534
Epoch: 14, Batch: 2000, Avg. Loss: 0.8111942699104547
Epoch: 14, Batch: 3000, Avg. Loss: 0.8052651465162635
Epoch: 15, Batch: 1000, Avg. Loss: 0.8006839917227626
Epoch: 15, Batch: 2000, Avg. Loss: 0.8126527510136365
Epoch: 15, Batch: 3000, Avg. Loss: 0.805940954118967
Epoch: 16, Batch: 1000, Avg. Loss: 0.8037247381657362
Epoch: 16, Batch: 2000, Avg. Loss: 0.8036714835092426
Epoch: 16, Batch: 3000, Avg. Loss: 0.7988248365819455
Epoch: 17, Batch: 1000, Avg. Loss: 0.7950976839363575
Epoch: 17, Batch: 2000, Avg. Loss: 0.8057148224562407
```

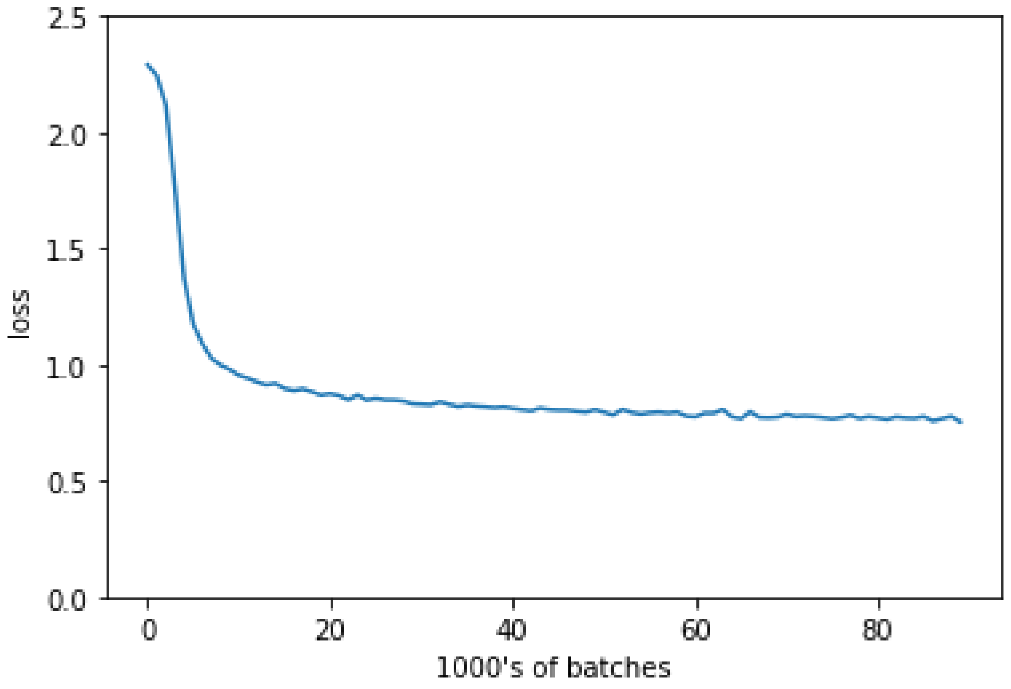
```
Epoch: 17, Batch: 3000, Avg. Loss: 0.7947029736340046
Epoch: 18, Batch: 1000, Avg. Loss: 0.7829015172049403
Epoch: 18, Batch: 2000, Avg. Loss: 0.8070014563947916
Epoch: 18, Batch: 3000, Avg. Loss: 0.7937779465019703
Epoch: 19, Batch: 1000, Avg. Loss: 0.7867941873371601
Epoch: 19, Batch: 2000, Avg. Loss: 0.792347573786974
Epoch: 19, Batch: 3000, Avg. Loss: 0.7951355692297221
Epoch: 20, Batch: 1000, Avg. Loss: 0.7909372003898024
Epoch: 20, Batch: 2000, Avg. Loss: 0.7952162316292525
Epoch: 20, Batch: 3000, Avg. Loss: 0.7778602090477943
Epoch: 21, Batch: 1000, Avg. Loss: 0.7748257178217173
Epoch: 21, Batch: 2000, Avg. Loss: 0.7907167854905128
Epoch: 21, Batch: 3000, Avg. Loss: 0.7908872455060482
Epoch: 22, Batch: 1000, Avg. Loss: 0.806639520265162
Epoch: 22, Batch: 2000, Avg. Loss: 0.7748053803294898
Epoch: 22, Batch: 3000, Avg. Loss: 0.7663869770616293
Epoch: 23, Batch: 1000, Avg. Loss: 0.7970829959958792
Epoch: 23, Batch: 2000, Avg. Loss: 0.7725547340661287
Epoch: 23, Batch: 3000, Avg. Loss: 0.7707921873256565
Epoch: 24, Batch: 1000, Avg. Loss: 0.7733363188952208
Epoch: 24, Batch: 2000, Avg. Loss: 0.7857834189385176
Epoch: 24, Batch: 3000, Avg. Loss: 0.7757597017213702
Epoch: 25, Batch: 1000, Avg. Loss: 0.7786088767498731
Epoch: 25, Batch: 2000, Avg. Loss: 0.77616107660532
Epoch: 25, Batch: 3000, Avg. Loss: 0.7716031022965908
Epoch: 26, Batch: 1000, Avg. Loss: 0.7658952854201198
Epoch: 26, Batch: 2000, Avg. Loss: 0.7721278666108846
Epoch: 26, Batch: 3000, Avg. Loss: 0.7817039262205362
Epoch: 27, Batch: 1000, Avg. Loss: 0.7677286018878221
Epoch: 27, Batch: 2000, Avg. Loss: 0.7765292583480478
Epoch: 27, Batch: 3000, Avg. Loss: 0.7700078958570957
Epoch: 28, Batch: 1000, Avg. Loss: 0.7630230665802956
Epoch: 28, Batch: 2000, Avg. Loss: 0.7752741254419089
Epoch: 28, Batch: 3000, Avg. Loss: 0.77007180082798
Epoch: 29, Batch: 1000, Avg. Loss: 0.7681610931977629
Epoch: 29, Batch: 2000, Avg. Loss: 0.7766445732414723
Epoch: 29, Batch: 3000, Avg. Loss: 0.7571060606241227
Epoch: 30, Batch: 1000, Avg. Loss: 0.7665402958467603
Epoch: 30, Batch: 2000, Avg. Loss: 0.7772204527258872
Epoch: 30, Batch: 3000, Avg. Loss: 0.7535791371688246
Finished Training
```

0.2 Visualizing the loss

A good indication of how much your network is learning as it trains is the loss over time. In this example, we printed and recorded the average loss for each 1000 batches and for each epoch. Let's plot it and see how the loss decreases (or doesn't) over time.

In this case, you can see that it takes a little bit for a big initial loss decrease, and the loss is flattening out over time.

```
In [13]: # visualize the loss as the network trained
         plt.plot(training_loss)
         plt.xlabel('1000\'s of batches')
         plt.ylabel('loss')
         plt.ylim(0, 2.5) # consistent scale
         plt.show()
```



0.2.1 Test the Trained Network

Once you are satisfied with how the loss of your model has decreased, there is one last step: test!

You must test your trained model on a previously unseen dataset to see if it generalizes well and can accurately classify this new dataset. For FashionMNIST, which contains many pre-processed training images, a good model should reach **greater than 85% accuracy** on this test dataset. If you are not reaching this value, try training for a larger number of epochs, tweaking your hyperparameters, or adding/subtracting layers from your CNN.

```
In [14]: # initialize tensor and lists to monitor test loss and accuracy
         test_loss = torch.zeros(1)
         class_correct = list(0. for i in range(10))
         class_total = list(0. for i in range(10))

         # set the module to evaluation mode
```

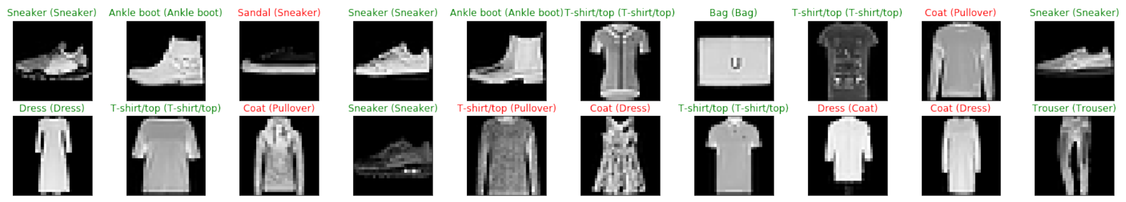
```
net.eval()

for batch_i, data in enumerate(test_loader):

    # get the input images and their corresponding labels
    inputs, labels = data

    # forward pass to get outputs
    outputs = net(inputs)

    # calculate the loss
    loss = criterion(outputs, labels)

    # update average test loss
    test_loss = test_loss + ((torch.ones(1) / (batch_i + 1)) * (loss.data - test_loss))

    # get the predicted class from the maximum value in the output-list of class scores
    _, predicted = torch.max(outputs.data, 1)

    # compare predictions to true label
    # this creates a `correct` Tensor that holds the number of correctly classified ima
    correct = np.squeeze(predicted.eq(labels.data.view_as(predicted)))

    # calculate test accuracy for *each* object class
    # we get the scalar value of correct items for a class, by calling `correct[i].item
    for i in range(batch_size):
        label = labels.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

print('Test Loss: {:.6f}\n'.format(test_loss.numpy()[0]))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))


print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 0.784023

Test Accuracy of T-shirt/top: 92% (925/1000)
Test Accuracy of Trouser: 96% (967/1000)
Test Accuracy of Pullover:  0% ( 0/1000)
Test Accuracy of Dress: 87% (873/1000)
Test Accuracy of  Coat: 91% (911/1000)
Test Accuracy of Sandal: 94% (945/1000)
Test Accuracy of Shirt:  0% ( 0/1000)
Test Accuracy of Sneaker: 93% (935/1000)
Test Accuracy of   Bag: 96% (967/1000)
Test Accuracy of Ankle boot: 93% (938/1000)

Test Accuracy (Overall): 74% (7461/10000)
```

0.2.2 Visualize sample test results

Format: predicted class (true class)

```
In [20]: # obtain one batch of test images
         dataiter = iter(test_loader)
         images, labels = dataiter.next()
         # get predictions
         preds = np.squeeze(net(images).data.max(1, keepdim=True)[1].numpy())
         images = images.numpy()

         # plot the images in the batch, along with predicted and true labels
         fig = plt.figure(figsize=(25, 4))
         for idx in np.arange(batch_size):
             ax = fig.add_subplot(2, batch_size/2, idx+1, xticks=[], yticks=[])
             ax.imshow(np.squeeze(images[idx]), cmap='gray')
             ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                          color=("green" if preds[idx]==labels[idx] else "red"))
```



0.2.3 Question: What are some weaknesses of your model? (And how might you improve these in future iterations.)

Answer: This model performs well on everything but shirts and pullovers (0% accuracy); it looks like this incorrectly classifies most of those as a coat which has a similar overall shape. Because it performs well on everything but these two classes, I suspect this model is overfitting certain classes at the cost of generalization. I suspect that this accuracy could be improved by adding some dropout layers to aoid overfitting.

```
In [21]: # Saving the model
         model_dir = 'saved_models/'
         model_name = 'fashion_net_simple.pt'

         # after training, save your model parameters in the dir 'saved_models'
         # when you're ready, un-comment the line below
         torch.save(net.state_dict(), model_dir+model_name)

In [ ]:
```