# Classify FashionMNIST, solution 2

May 27, 2020

## 0.1 # CNN for Classification

In this notebook, we define **and train** an 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 second solution will be a CNN with two convolutional layers **and** additional fully-connected and dropout layers to avoid over-fitting the data and gradient descent with momentum to avoid reaching a local minimum. The batch size and number of epochs to train are the same as in the first example solution so that you can see how the structure of the network and loss hyperparameters have affected the accuracy of the model!

Again, 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))
```

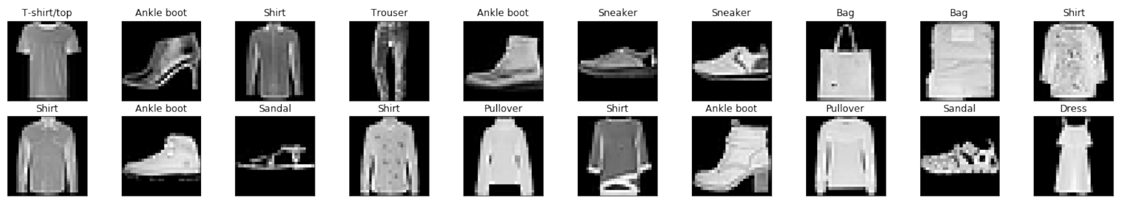
```
Train data, number of images:  60000
Test data, number of images:  10000
```

```
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 in either the `__init__` or 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
                self.fc1 = nn.Linear(20*5*5, 50)

                # dropout with p=0.4
                self.fc1_drop = nn.Dropout(p=0.4)

                # finally, create 10 output channels (for the 10 classes)
                self.fc2 = nn.Linear(50, 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
                # this line of code is the equivalent of Flatten in Keras
                x = x.view(x.size(0), -1)

                # two linear layers with dropout in between
                x = F.relu(self.fc1(x))
                x = self.fc1_drop(x)
                x = self.fc2(x)

                # 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=50, bias=True)
  (fc1_drop): Dropout(p=0.4)
  (fc2): Linear(in_features=50, 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()`. 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
        # using cross entropy whcih combines softmax and NLL loss
        criterion = nn.CrossEntropyLoss()
```

```
## TODO: specify optimizer
# stochastic gradient descent with a small learning rate AND some momentum
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

### 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:  9.75
```

### 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
training_loss = train(n_epochs)
```

```
Epoch: 1, Batch: 1000, Avg. Loss: 1.5503497277498246
Epoch: 1, Batch: 2000, Avg. Loss: 0.9823762179017067
Epoch: 1, Batch: 3000, Avg. Loss: 0.8715841763317586
Epoch: 2, Batch: 1000, Avg. Loss: 0.7868324071615934
Epoch: 2, Batch: 2000, Avg. Loss: 0.740295095950365
Epoch: 2, Batch: 3000, Avg. Loss: 0.7027995853126049
Epoch: 3, Batch: 1000, Avg. Loss: 0.6736214982271195
Epoch: 3, Batch: 2000, Avg. Loss: 0.6420598052889108
Epoch: 3, Batch: 3000, Avg. Loss: 0.6165065146535635
Epoch: 4, Batch: 1000, Avg. Loss: 0.6010752868279815
Epoch: 4, Batch: 2000, Avg. Loss: 0.5861366413384675
Epoch: 4, Batch: 3000, Avg. Loss: 0.5723311684802175
Epoch: 5, Batch: 1000, Avg. Loss: 0.556285381063819
Epoch: 5, Batch: 2000, Avg. Loss: 0.5371940165311098
Epoch: 5, Batch: 3000, Avg. Loss: 0.540051191739738
Epoch: 6, Batch: 1000, Avg. Loss: 0.5274158929437399
Epoch: 6, Batch: 2000, Avg. Loss: 0.5127518430128694
Epoch: 6, Batch: 3000, Avg. Loss: 0.5032681580260396
Epoch: 7, Batch: 1000, Avg. Loss: 0.4870288127809763
Epoch: 7, Batch: 2000, Avg. Loss: 0.505244158513844
Epoch: 7, Batch: 3000, Avg. Loss: 0.496015349663794
Epoch: 8, Batch: 1000, Avg. Loss: 0.48576754093915225
Epoch: 8, Batch: 2000, Avg. Loss: 0.48350561733543873
Epoch: 8, Batch: 3000, Avg. Loss: 0.47196527863666415
Epoch: 9, Batch: 1000, Avg. Loss: 0.466754915881902
Epoch: 9, Batch: 2000, Avg. Loss: 0.47388256136327983
Epoch: 9, Batch: 3000, Avg. Loss: 0.4644597099982202
Epoch: 10, Batch: 1000, Avg. Loss: 0.45841106051206587
Epoch: 10, Batch: 2000, Avg. Loss: 0.4538198432326317
Epoch: 10, Batch: 3000, Avg. Loss: 0.4536016509756446
Epoch: 11, Batch: 1000, Avg. Loss: 0.45032810129970313
Epoch: 11, Batch: 2000, Avg. Loss: 0.4384474340081215
Epoch: 11, Batch: 3000, Avg. Loss: 0.4525843020901084
Epoch: 12, Batch: 1000, Avg. Loss: 0.4386790323778987
Epoch: 12, Batch: 2000, Avg. Loss: 0.4385225551202893
Epoch: 12, Batch: 3000, Avg. Loss: 0.4379486278668046
Epoch: 13, Batch: 1000, Avg. Loss: 0.4338793611228466
Epoch: 13, Batch: 2000, Avg. Loss: 0.4356810339614749
Epoch: 13, Batch: 3000, Avg. Loss: 0.4249481642395258
Epoch: 14, Batch: 1000, Avg. Loss: 0.4244724450223148
Epoch: 14, Batch: 2000, Avg. Loss: 0.42580091661959885
Epoch: 14, Batch: 3000, Avg. Loss: 0.4134297588169575
Epoch: 15, Batch: 1000, Avg. Loss: 0.41352825256437065
Epoch: 15, Batch: 2000, Avg. Loss: 0.41216779660433533
```

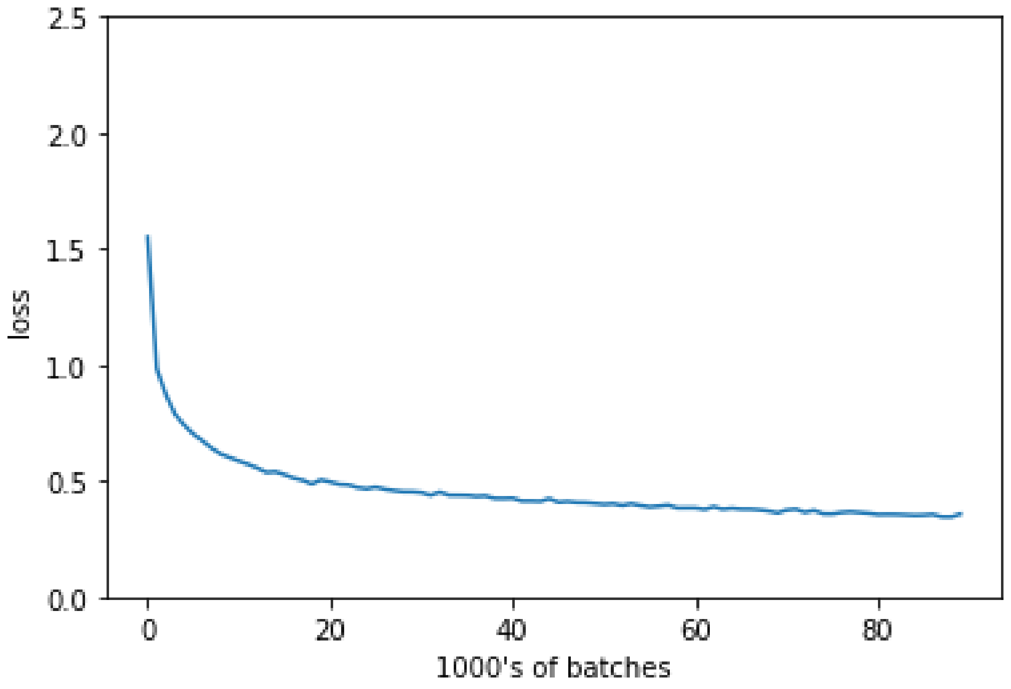
```
Epoch: 15, Batch: 3000, Avg. Loss: 0.4238966451883316
Epoch: 16, Batch: 1000, Avg. Loss: 0.40851068092137577
Epoch: 16, Batch: 2000, Avg. Loss: 0.41172413343563674
Epoch: 16, Batch: 3000, Avg. Loss: 0.407512312008068
Epoch: 17, Batch: 1000, Avg. Loss: 0.4069803262837231
Epoch: 17, Batch: 2000, Avg. Loss: 0.40323004772514104
Epoch: 17, Batch: 3000, Avg. Loss: 0.39955010274797675
Epoch: 18, Batch: 1000, Avg. Loss: 0.4021722289249301
Epoch: 18, Batch: 2000, Avg. Loss: 0.3949743340909481
Epoch: 18, Batch: 3000, Avg. Loss: 0.40229857332631946
Epoch: 19, Batch: 1000, Avg. Loss: 0.3943412540536374
Epoch: 19, Batch: 2000, Avg. Loss: 0.38810314409434793
Epoch: 19, Batch: 3000, Avg. Loss: 0.3916338127143681
Epoch: 20, Batch: 1000, Avg. Loss: 0.39640154411271217
Epoch: 20, Batch: 2000, Avg. Loss: 0.38369314727559684
Epoch: 20, Batch: 3000, Avg. Loss: 0.38399071975797416
Epoch: 21, Batch: 1000, Avg. Loss: 0.3839878745302558
Epoch: 21, Batch: 2000, Avg. Loss: 0.37751691177859903
Epoch: 21, Batch: 3000, Avg. Loss: 0.3890007802695036
Epoch: 22, Batch: 1000, Avg. Loss: 0.3783960998430848
Epoch: 22, Batch: 2000, Avg. Loss: 0.38276130944490433
Epoch: 22, Batch: 3000, Avg. Loss: 0.37880005171522496
Epoch: 23, Batch: 1000, Avg. Loss: 0.3786282137185335
Epoch: 23, Batch: 2000, Avg. Loss: 0.37593254062533377
Epoch: 23, Batch: 3000, Avg. Loss: 0.37070120150595903
Epoch: 24, Batch: 1000, Avg. Loss: 0.3627518704533577
Epoch: 24, Batch: 2000, Avg. Loss: 0.37503556187823417
Epoch: 24, Batch: 3000, Avg. Loss: 0.37880107753351333
Epoch: 25, Batch: 1000, Avg. Loss: 0.36501296575367453
Epoch: 25, Batch: 2000, Avg. Loss: 0.3736507741995156
Epoch: 25, Batch: 3000, Avg. Loss: 0.35865319488197567
Epoch: 26, Batch: 1000, Avg. Loss: 0.357891530200839
Epoch: 26, Batch: 2000, Avg. Loss: 0.36472464990988374
Epoch: 26, Batch: 3000, Avg. Loss: 0.3674461318999529
Epoch: 27, Batch: 1000, Avg. Loss: 0.3642365995682776
Epoch: 27, Batch: 2000, Avg. Loss: 0.36075587628036737
Epoch: 27, Batch: 3000, Avg. Loss: 0.35526194078847767
Epoch: 28, Batch: 1000, Avg. Loss: 0.35584388167224823
Epoch: 28, Batch: 2000, Avg. Loss: 0.35545510455779733
Epoch: 28, Batch: 3000, Avg. Loss: 0.353847408330068
Epoch: 29, Batch: 1000, Avg. Loss: 0.3518907689526677
Epoch: 29, Batch: 2000, Avg. Loss: 0.3528207621723414
Epoch: 29, Batch: 3000, Avg. Loss: 0.35581499799340965
Epoch: 30, Batch: 1000, Avg. Loss: 0.3452995559573174
Epoch: 30, Batch: 2000, Avg. Loss: 0.34490807877480983
Epoch: 30, Batch: 3000, Avg. Loss: 0.3573662136420608
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 should see that the loss has an initially large decrease and even looks like it would decrease more (by some small, linear amount) if we let it train for more epochs.

```
In [19]: # 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 [20]: # initialize tensor and lists to monitor test loss and accuracy
         test_loss = torch.zeros(1)
```

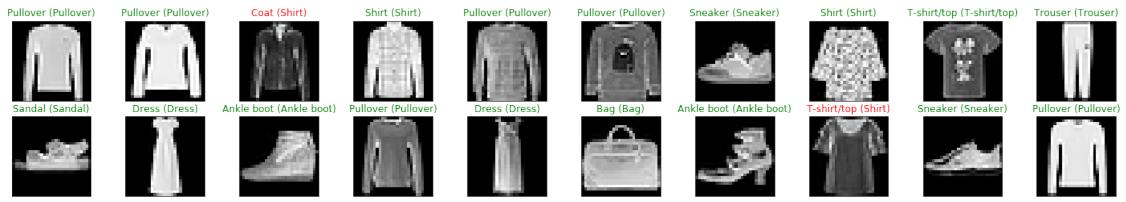
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

# 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: 0.326777

Test Accuracy of T-shirt/top: 88% (880/1000)
Test Accuracy of Trouser: 97% (970/1000)
Test Accuracy of Pullover: 83% (837/1000)
Test Accuracy of Dress: 90% (901/1000)
Test Accuracy of  Coat: 75% (758/1000)
Test Accuracy of Sandal: 96% (965/1000)
Test Accuracy of Shirt: 57% (576/1000)
Test Accuracy of Sneaker: 96% (967/1000)
Test Accuracy of   Bag: 97% (977/1000)
Test Accuracy of Ankle boot: 95% (950/1000)

Test Accuracy (Overall): 87% (8781/10000)
```

### 0.2.2 Visualize sample test results

Format: predicted class (true class)

```
In [25]: # 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**: Since t-shirts, shirts, and coats have a similar overall shape, my model has trouble distinguishing between those items. In fact, its lowest test class accuracy is: `Test Accuracy of Shirt,` which this model only gets right about 60% of the time .

I suspect that this accuracy could be improved by doing some data augmentation with respect to these classes or even adding another convolutional layer to extract even higher level features.

```
In [26]: # Saving the model
         model_dir = 'saved_models/'
         model_name = 'fashion_net_ex.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 [ ]:
```