Classify FashionMNIST, exercise

July 6, 2018

0.1 # CNN for Classification

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

0.1.1 Load the data

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

In [8]:
```
# 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 [9]: # 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

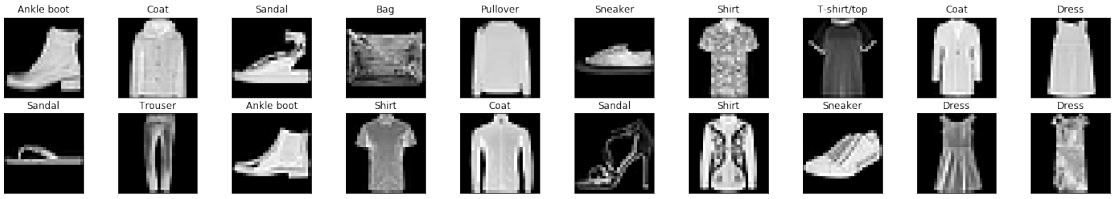
This cell iterates over the training dataset, loading a random batch of image/label data, using `dataiter.next()`. It then plots the batch of images and labels in a `2 x batch_size/2` grid.

```
In [10]: 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` functio; 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 convoluttional/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

In [22]:
```
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)
)
```

0.1.5 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 [25]: import torch.optim as optim

        ## TODO: specify loss function (try categorical cross-entropy)
        criterion = nn.CrossEntropyLoss()

        ## TODO: specify optimizer
        optimizer = optim.SGD(net.parameters(), lr=0.001)
```

0.1.6 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).

Variable Before an input tensor can be processed by a model, it must be wrapped in a Variable wrapper; this wrapper gives PyTorch the ability to automatically track how this input changes as it passes through the network and automatically calculate the gradients needed for backpropagation.

```
In [26]: from torch.autograd import Variable

        # 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:  10.99
```

0.1.7 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 training dataset.

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

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

```
In [27]: from torch.autograd import Variable

        def train(n_epochs):

            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

                    # wrap them in a torch Variable
```

```
                    inputs, labels = Variable(inputs), Variable(labels)

                    # 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
                    running_loss += loss.data[0]
                    if batch_i % 1000 == 999:    # print every 1000 mini-batches
                        print('Epoch: {}, Batch: {}, Avg. Loss: {}'.format(epoch + 1, batch_i+1
                        running_loss = 0.0

            print('Finished Training')
```

In [28]:
```
# define the number of epochs to train for
n_epochs = 5 # start small to see if your model works, initially

# call train
train(n_epochs)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:32: UserWarning: invalid index of a


Epoch: 1, Batch: 1000, Avg. Loss: 2.2983899116516113
Epoch: 1, Batch: 2000, Avg. Loss: 2.2843172550201416
Epoch: 1, Batch: 3000, Avg. Loss: 2.2572150230407715
Epoch: 2, Batch: 1000, Avg. Loss: 2.1787383556365967
Epoch: 2, Batch: 2000, Avg. Loss: 1.9909937381744385
Epoch: 2, Batch: 3000, Avg. Loss: 1.8168822526931763
Epoch: 3, Batch: 1000, Avg. Loss: 1.7088547945022583
Epoch: 3, Batch: 2000, Avg. Loss: 1.6200265884399414
Epoch: 3, Batch: 3000, Avg. Loss: 1.5745861530303955
Epoch: 4, Batch: 1000, Avg. Loss: 1.537441611289978
Epoch: 4, Batch: 2000, Avg. Loss: 1.4384881258010864
Epoch: 4, Batch: 3000, Avg. Loss: 1.4008946418762207
Epoch: 5, Batch: 1000, Avg. Loss: 1.3720191717147827
```

```
Epoch: 5, Batch: 2000, Avg. Loss: 1.3582733869552612
Epoch: 5, Batch: 3000, Avg. Loss: 1.338188648223877
Finished Training
```

0.1.8 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 [30]: # 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

             # wrap them in a torch Variable
             # volatile means we do not have to track how the inputs change
             inputs, labels = Variable(inputs, volatile=True), Variable(labels, volatile=True)

             # 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
             correct = np.squeeze(predicted.eq(labels.data.view_as(predicted)))

             # calculate test accuracy for *each* object class
             for i in range(batch_size):
```

```
            label = labels.data[i]
            class_correct[label] += correct[i]
            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)))
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:16: UserWarning: volatile was remov
  app.launch_new_instance()


Test Loss: 1.345740
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-30-c815a6b7e8d5> in <module>()
 42     if class_total[i] > 0:
 43         print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
---> 44             classes[i], 100 * class_correct[i] / class_total[i],
 45             np.sum(class_correct[i]), np.sum(class_total[i])))
 46     else:


RuntimeError: value cannot be converted to type uint8_t without overflow: 1000.000000
```

0.1.9 Visualize sample test results

In [ ]:
```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()
# get predictions
```

```
preds = np.squeeze(net(Variable(images, volatile=True)).data.max(1, keepdim=True)[1].num
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"))
```

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

Answer: Double-click and write answer, here.

0.1.11 Save Your Best Model

Once you've decided on a network architecture and are satisfied with the test accuracy of your model after training, it's time to save this so that you can refer back to this model, and use it at a later data for comparison of for another classification task!

In [ ]:
```
## TODO: change the model_name to something uniqe for any new model
## you wish to save, this will save it in the saved_models directory
model_dir = 'saved_models/'
model_name = 'model_1.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)
```

0.1.12 Load a Trained, Saved Model

To instantiate a trained model, you'll first instantiate a new `Net()` and then initialize it with a saved dictionary of parameters (from the save step above).

In [ ]:
```
# instantiate your Net
# this refers to your Net class defined above
net = Net()

# load the net parameters by name
# uncomment and write the name of a saved model
#net.load_state_dict(torch.load('saved_models/model_1.pt'))

print(net)

# Once you've loaded a specific model in, you can then
# us it or further analyze it!
# This will be especialy useful for feature visualization
```