1 - Fully Connected Network from scratch

1.1 Loading the FashionMNIST dataset

Download the dataset

The source comes from Fashion-MNIST https://github.com/zalandoresearch/fashion-mnist

Note: If dataset is already downloaded, it is not downloaded again.

```
import torchvision

transform = torchvision.transforms.Compose(
    [torchvision.transforms.ToTensor(), torchvision.transforms.Normalize(
        (0.5,0.5,0.5), (0.5,0.5,0.5))])
trainset = torchvision.datasets.FashionMNIST(root='./data', train=True,
                                             download=True, transform=transform)
testset = torchvision.datasets.FashionMNIST(root='./data', train=False,
                                            download=True, transform=transform)
```

Pre-processing the dataset

Get the training dataset.

```
X_train = trainset.data
print(X_train.shape)
```

```
torch.Size([60000, 28, 28])
```

Flatten (60000,28,28) size of X_train into (784,60000) size of matrix.

Also, normalize each pixel value to between 0 and 1.

```
import numpy as np
X_train = X_train.reshape(X_train.shape[0], -1).T / 255.0
print(X_train.shape)
```

Get the testing dataset

```
X_test = testset.data
print(X_test.shape)
```

Flatten (10000,28,28) size of X_test into (784,10000) size of matrix.

Also, normalize each pixel value to between 0 and 1.

```
X_test = X_test.reshape(X_test.shape[0], -1).T / 255.0
print(X_test.shape)
```

Get the label from training dataset.

```
y_train = trainset.targets
print(y_train)
```

Transfer training label from tensor to numpy.

```
y_train = np.reshape(np.asarray(y_train), (60000,))
print(y_train.shape)
```

Get the label from testing dataset.

And, transfer it into numpy.

```
y_test = testset.targets
y_test = np.reshape(np.asarray(y_test), (10000,))
print(y_test.shape)
```

This is the label class:

```
label = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
```

```
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')
```

Let's take a look of the first two picture we get in the training dataset:

```
fp1 = X_train[:,0]
fp2 = X_train[:,1]

import matplotlib.pyplot as plt
f, ax = plt.subplots(1,2)

ax[0].imshow(fp1.reshape((28,28)), cmap="gray") #first image
ax[1].imshow(fp2.reshape((28,28)), cmap="gray") #second image
plt.show()
```

The label of the training dataset shown above, the first picture is index 9 and second picture is index 0.

As we can see in the label class, index 9 is 'Ankle Boot', index 0 is 'T-shirt/top'.

Shuffle the dataset

Shuffle the training and validation dataset, with label, by index.

```
def split(X, y, val_ratio=0.1):
    val_number = int(val_ratio * X.shape[1])
    random_indice = np.random.permutation(X.shape[1])
    return X[:, random_indice[val_number:]],y[random_indice[val_number:]], \
           X[:, random_indice[:val_number]], \
           y[random_indice[:val_number]]

X_train, y_train, X_val, y_val = split(X_train, y_train)
```

One Hot Encoding Labels

Make one dimension labels from (dims,) to (10, dims) for the better prediction of the Neural Network.

```
def one_hot_encoding(label):
    shape = (label.max() + 1, label.size)
    rows = np.arange(label.size)
    one_hot = np.zeros(shape)
    one_hot[label, rows] = 1
    return one_hot

y_train = one_hot_encoding(y_train)
y_val = one_hot_encoding(y_val)
y_test = one_hot_encoding(y_test)

print(y_test.shape)
```

1.2 Implementing the Network

```
class NeuralNetwork:
    def __init__(self, layer_dims, seed_value=99):
        """
        Arguments:
        layer_dims -- A list contains the dimensions of each layer in CNN
        seed_value -- set seed to generate random numpy

        Attributes generated:
        parameters -- a dict contains parameters "W1", "b1", ..., "WL",
                      "bL"of each corresponding layer
                    Wl -- weight matrix of shape (layer_dims[l],layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
        nn_architecture -- the dictionary contains the description of the
                           network architecture
                    'input_dim'  -- input dimension from last network layer
                    'output_dim' -- output dimension after applying the
                                    activation
                    'activation' -- ReLU or Softmax
        X_val -- validation data in training
        y_val -- validation label in training
        cost_history -- stores cross entrophy loss for every training iteration
        acc_history -- stores the training accuracy for every iteration
        """

        np.random.seed(seed_value)
        self.parameters = {}
        self.nn_architecture = []
```

```
        self.X_val = None
        self.y_val = None
        self.cost_history = []
        self.acc_history = []

        input_dim = layer_dims[0]
        for idx, output_dim in enumerate(layer_dims[1:]):
            self.nn_architecture.append({'input_dim': input_dim, 'output_dim': \
                                         output_dim, 'activation': 'relu'})
            input_dim = output_dim
        self.nn_architecture[len(self.nn_architecture) - 1]['activation'] = \
                                                            'softmax'

        for idx, layer in enumerate(self.nn_architecture):
            layer_idx = idx + 1
            layer_input_size = layer["input_dim"]
            layer_output_size = layer["output_dim"]

            self.parameters['W' + str(layer_idx)] = np.random.randn(
                layer_output_size, layer_input_size) * 0.1
            self.parameters['b' + str(layer_idx)] = np.random.randn(
                layer_output_size, 1) * 0.1

    def relu(self, Z):
        """
        Implement the ReLU function

        Arguments:
        Z -- output of linear portion of the output layer (last layer)

        Returns:
        Rectified linear units
        """
        return np.maximum(0, Z)

    def softmax(self, Z):
        """
        Implement the Softmax function

        Arguments:
        Z -- output of linear portion of the output layer (last layer)

        Returns:
        p -- softmax probability
        """
        p = np.exp(Z - np.max(Z, axis=0, keepdims=True))
        p /= np.sum(p, axis=0, keepdims=True)
        return p

    def affineForward(self, A, W, b, activation="relu"):
        """
        Implement the linear portion of CNN's forward propagation
```

```
        Implement the linear portion of CNN's forward propagation.

        Arguments:
        A -- activation from previous layer
        W -- weights matrix
        b -- bias vector

        Returns:
        The neural layer after applied activation function
        Z -- the input of the activation function
        """
        Z = np.dot(W, A) + b

        if activation == "relu":
            activation_func = self.relu
        elif activation == "softmax":
            activation_func = self.softmax
        else:
            raise Exception('Non-supported activation function')

        return activation_func(Z), Z

    def forwardPropagation(self, X):
        """
        Implement the forward propagation

        Arguments:
        X -- input from input layer as the starting point of forward propagation

        Returns:
        A_curr -- the output of the activation function after each layer
        cache -- a list stores parameters (cache) for each layer during
                 forward propagation
        """
        A_curr = X
        cache = {}

        for idx, layer in enumerate(self.nn_architecture):
            layer_idx = idx + 1
            A_prev = A_curr

            activ_function_curr = layer["activation"]
            W_curr = self.parameters["W" + str(layer_idx)]
            b_curr = self.parameters["b" + str(layer_idx)]
            A_curr, Z_curr = self.affineForward(A_prev, W_curr, b_curr,
                                                activ_function_curr)

            cache["A" + str(idx)] = A_prev
            cache["Z" + str(layer_idx)] = Z_curr

        return A_curr, cache
```

```
    def costFunction(self, AL, y, epsilon=1e-12):
        """
        Implement the cross entropy loss

        Arguments:
        AL -- output of linear portion of the output layer (last layer)
        y -- the labels of data
        epsilon -- a very small value, in case the AL is zero when log

        Returns:
        cost -- the cross entropy loss value of each iteration of forward
                propagation
        """
        cost = -np.mean(np.multiply(y, np.log(AL + epsilon)))
        return cost

    def derivative_cost(self, AL, y):
        """
        Implement the first step of back propagation: the derivative of cost
        function over AL

        Arguments:
        AL -- output of linear portion of the output layer (last layer)
        y -- the labels of data

        Returns:
        dAL -- the derivative of cost function over AL
        """
        dAL = AL - y
        return dAL

    def affineBackward(self,dA_curr,W_curr,Z_curr,A_prev,activation="relu"):
        """
        Implement the linear portion of backward propagation of one layer(1)

        Arguments:
        dA_curr -- Gradient of the cost over activation output from current layer (layer l-1)
        W_curr -- W for the current layer(layerl)
        Z_curr -- Z for the current layer(layerl)
        A_prev -- A for the previous layer(layerl)

        Returns:
        dA_prev -- Gradient of the cost over activation output from previous layer (layer l-1
        dW_curr -- Gradient of the cost over W for the current layer(layerl)
        db_curr -- Gradient of the cost over b for the current layer(layer1)
        """
        if activation == "relu":
            dZ_curr = self.derivative_relu(dA_curr, Z_curr)
        elif activation == "softmax":
            dZ_curr = dA_curr
        else:
            raise Exception('Non-supported activation function')
```

```
            raise Exception('Non-supported activation function')

        m = A_prev.shape[1]
        dW_curr = np.dot(dZ_curr, A_prev.T) / m
        db_curr = np.sum(dZ_curr, axis=1, keepdims=True) / m
        dA_prev = np.dot(W_curr.T, dZ_curr)

        return dA_prev, dW_curr, db_curr

    def derivative_relu(self, dA, Z):
        """
        Implement the derivative calculation of relu activation during backpropagation

        Arguments:
        dA -- the gradient of cost function over relu activation ouput A

        Returns:
        dZ -- the derivative of cost function over Z
        """
        dZ = np.array(dA, copy=True)
        dZ[Z <= 0] = 0
        return dZ

    def backPropagation(self, dAL, cache):
        """
        Implement backpropagation for each layer

        Arguments:
        dAL -- the gradient of cost function over the activation output of last
               layer: starting point for backpropagation
        cache -- a list stores tuple of A,W,b,Z for each layer

        Returns:
        grads -- a dict stores dWl and dbl for each layer l
        """
        grads = {}

        # initiation of backward propagation
        dA_prev = dAL

        for layer_idx_prev, layer in reversed(list(enumerate(
                                              self.nn_architecture))):
            # we number network layers from 1
            layer_idx_curr = layer_idx_prev + 1
            # extraction of the activation function for the current layer
            activ_function_curr = layer["activation"]
            # update dA
            dA_curr = dA_prev
            # get variables
            A_prev = cache["A" + str(layer_idx_prev)]
            Z_curr = cache["Z" + str(layer_idx_curr)]
            W_curr = self.parameters["W" + str(layer_idx_curr)]
```

```
            # get the gradient of the cost
            dA_prev, dW_curr, db_curr = self.affineBackward(
                dA_curr, W_curr, Z_curr, A_prev, activ_function_curr)
            # update gradient values
            grads["dW" + str(layer_idx_curr)] = dW_curr
            grads["db" + str(layer_idx_curr)] = db_curr

        return grads

    def updateParameters(self, grads, alpha):
        """
        Use gradient descent to implement updated parameters

        Arguments:
        grads -- a dict stores all parameters gradients for each layer
        alpha -- learning rate
        """
        for layer_idx, layer in enumerate(self.nn_architecture, 1):
            self.parameters["W" + str(layer_idx)] -= alpha * grads["dW" + \
                                                     str(layer_idx)]
            self.parameters["b" + str(layer_idx)] -= alpha * grads["db" + \
                                                     str(layer_idx)]

    def train(self,X,y,iters,alpha,batch_size,verbose=False,print_every=100):
        """
        It takes advantage of every function in this class to implement training
        and validation using CNN.

        Arguments:
        X -- input data
        y -- labels of data
        iters -- number of iterations to run
        alpha -- learning rate
        batch_size -- number of samples to assign to minibatch
        verbose -- decide whether to print or not print loss and accuracy
        print_every -- number of iterations to print

        Return:
        Numbewr of iterations, train loss, train_acc, and valid_acc in every
        customized iterations
        """
        for i in range(0, iters):
            X_batch, y_batch = self.get_batch(X, y, batch_size)
            AL, cache = self.forwardPropagation(X_batch)
            loss = self.costFunction(AL, y_batch)
            dAL = self.derivative_cost(AL, y_batch)
            grads = self.backPropagation(dAL, cache)
            self.updateParameters(grads, alpha)
            train_acc = self.score(self.predict(X), y)
            if verbose:
                if i == 0 or (i+1) % print_every == 0:
                    val_acc = self.score(self.predict(self.X_val), self.y_val)
```

```
                print('iter={:4}, loss={:.6f}, train_acc={:.6f}, validation_acc={:.6f}'.f
            self.cost_history.append(loss)
            self.acc_history.append(train_acc)

    def predict(self, X):
        """
        It predicts the label given input x
        Argument:
        X -- input of data
        Return:
        y_pred -- predicted label
        """
        AL, _ = self.forwardPropagation(X)
        y_pred = np.argmax(AL, axis=0)
        return y_pred

    def score(self, y_pred, y_true):
        """
        It calculates the percentage of correct predicted labels over true labels

        Argument:
        y_pred-- predicted labels
        y_true -- true labels

        REturn: percentage of correct predicted labels
        """
        target = np.argmax(y_true, axis=0)
        correct = np.mean(y_pred == target)
        return correct

    def load_validation_set(self, X_val, y_val):
        """
        Load validation set to CNN
        """
        self.X_val = X_val
        self.y_val = y_val

    def get_batch(self, X, y, batch_size):
        """
        Load mini-batch to CNN
        """
        batch_index = np.random.randint(X.shape[1], size=batch_size)
        X_batch = X[:, batch_index]
        y_batch = y[:, batch_index]
        return X_batch, y_batch
```

1.3 Predicting

Define variables for training the Fully Connected Neural Network:

```
batch_size = 1000
seed_value = 2
learning_rate = 0.1
iteration = 120
print_every = 10
layer_dims = [X_train.shape[0], 1024, 256, 128, 10]
```

Train the F-CNN:

```
CNN = NeuralNetwork(layer_dims, seed_value)
CNN.load_validation_set(X_val, y_val)
CNN.train(X_train, y_train, iteration, learning_rate, batch_size, True, print_every)
```

```
iter=   1, loss=0.429308, train_acc=0.100111, validation_acc=0.099167
iter=  10, loss=0.166481, train_acc=0.461574, validation_acc=0.455000
iter=  20, loss=0.101982, train_acc=0.656241, validation_acc=0.646500
iter=  30, loss=0.096968, train_acc=0.680870, validation_acc=0.676167
iter=  40, loss=0.071267, train_acc=0.709759, validation_acc=0.699333
iter=  50, loss=0.070132, train_acc=0.763889, validation_acc=0.761167
iter=  60, loss=0.067600, train_acc=0.742463, validation_acc=0.736667
iter=  70, loss=0.061559, train_acc=0.771593, validation_acc=0.767667
iter=  80, loss=0.059510, train_acc=0.788759, validation_acc=0.784000
iter=  90, loss=0.051820, train_acc=0.806685, validation_acc=0.804333
iter= 100, loss=0.068209, train_acc=0.765444, validation_acc=0.759833
iter= 110, loss=0.053147, train_acc=0.820815, validation_acc=0.819500
iter= 120, loss=0.063519, train_acc=0.789130, validation_acc=0.785333
```

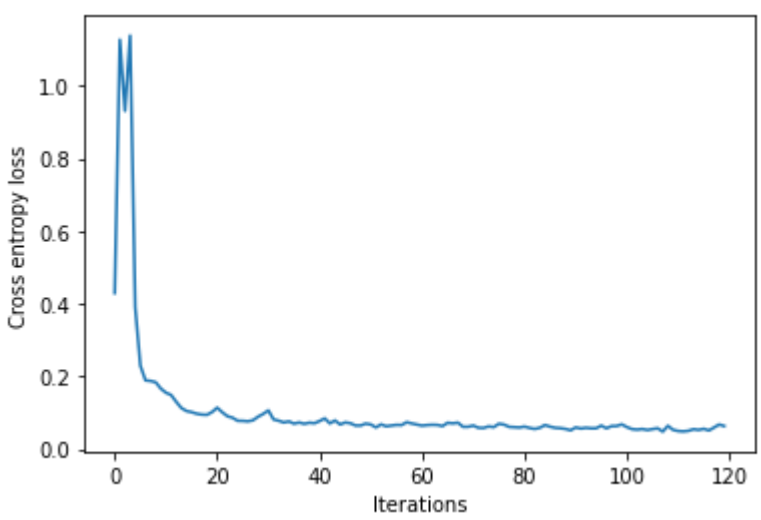
Test the accuracy based on the testing dataset:

```
y_pred = CNN.predict(X_test)
test_acc = CNN.score(y_pred, y_test)
print('Accuracy pridicting test dataset = {:.2%}'.format(test_acc))
```

```
Accuracy pridicting test dataset = 77.97%
```

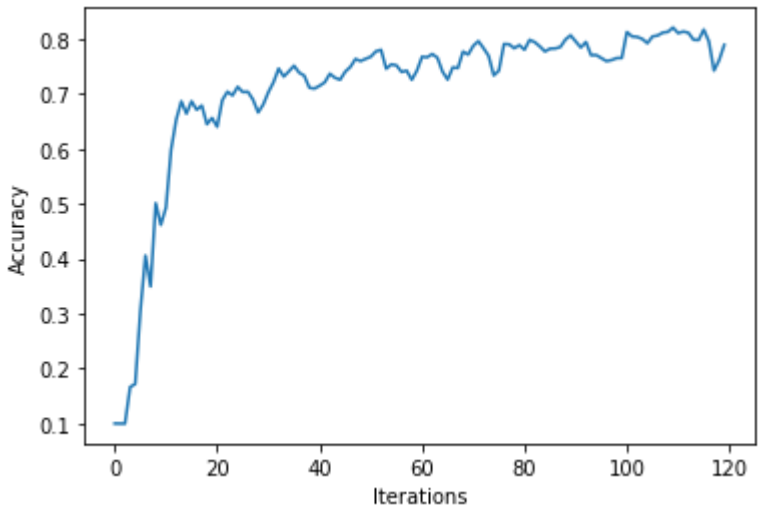
To visualize the performance of the Neural Network, a plot below shows the cross entropy loss during the iterations:

```
import matplotlib.pyplot as plt
plt.plot(CNN.cost_history)
plt.xlabel('Iterations')
plt.ylabel('Cross entropy loss')
plt.show()
```



Below shows the prediction accuracy of the training data:

```
plt.plot(CNN.acc_history)
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.show()
```



2 - CNN using PyTorch

Libraries used:

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
```

```
import torchvision
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
```

2.1 Loading the FashionMNIST dataset

Download the dataset

```
trainSet = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True,
                                       transform=transforms.ToTensor())
testSet = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True,
                                       transform=transforms.ToTensor())
```

Make sure you turn on the GPU in Runtime setting, for a better performance.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Get the train/test loader using DataLoader

With batch size --> 100, and Shuffle is TRUE.

```
trainLoader = torch.utils.data.DataLoader(trainSet,batch_size=100,shuffle=True)
testLoader = torch.utils.data.DataLoader(testSet,batch_size=100,shuffle=True)
```

Creating a method to name the class for the label.

example: 9 --> Ankle Boot

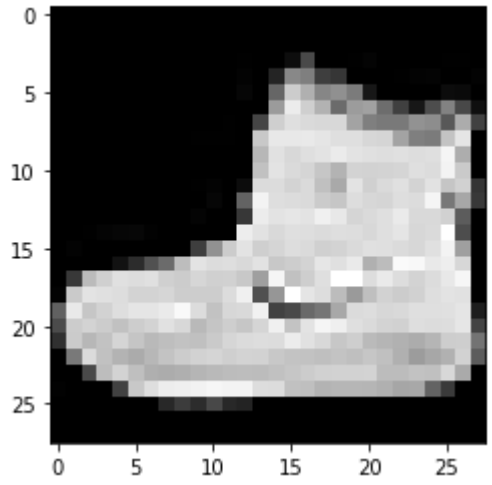
```
def outputLabel(label):
    outputMapping = {
                 0: "T-shirt/Top",
                 1: "Trouser",
                 2: "Pullover",
                 3: "Dress",
                 4: "Coat",
                 5: "Sandal",
                 6: "Shirt",
                 7: "Sneaker",
```

```
                8: "Bag",
                9: "Ankle Boot"
                }
    input = (label.item() if type(label) == torch.Tensor else label)
    return outputMapping[input]
```

- To show a data using matplotlib :

```
image, label = next(iter(trainSet))
plt.imshow(image.squeeze(), cmap="gray")
print(label)
```

```
9
```



2.2 Implementing the Network

- Building the CNN class:

  Model Class Name: FashionCNN

  Layers: 2 sequential layers consist of:

```
        *  Convolution layer with kernal -> 3*3, padding = 1 (1st layer) & padding = 0 (2nd layer).
        *  Stride of 1 in both the layer
        *  Activation function: Relu
        *  All the functionaltiy is given in forward method that defines the forward pass of CNN.
        *  Output:
              1.  1st Conv later:    input: 28 * 28 * 3   and  Output : 28 * 28 * 32.
              2.  Max pooling layer: input: 28 * 28 * 32  and Output:  14 * 14 * 32.
              3.  2nd Conv layer:    input : 14 * 14 * 32 and output: 12 * 12 * 64
              4.  2nd Max Pooling layer : 12 * 12 * 64, output: 6 * 6 * 64.
```

At last fully connected layer has 10 output features for 10 types of clothes.

```
class FashionCNN(nn.Module):

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

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.fc1 = nn.Linear(in_features=64*6*6, out_features=600)
        self.drop = nn.Dropout2d(0.25)
        self.fc2 = nn.Linear(in_features=600, out_features=120)
        self.fc3 = nn.Linear(in_features=120, out_features=10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.drop(out)
        out = self.fc2(out)
        out = self.fc3(out)

        return out
```

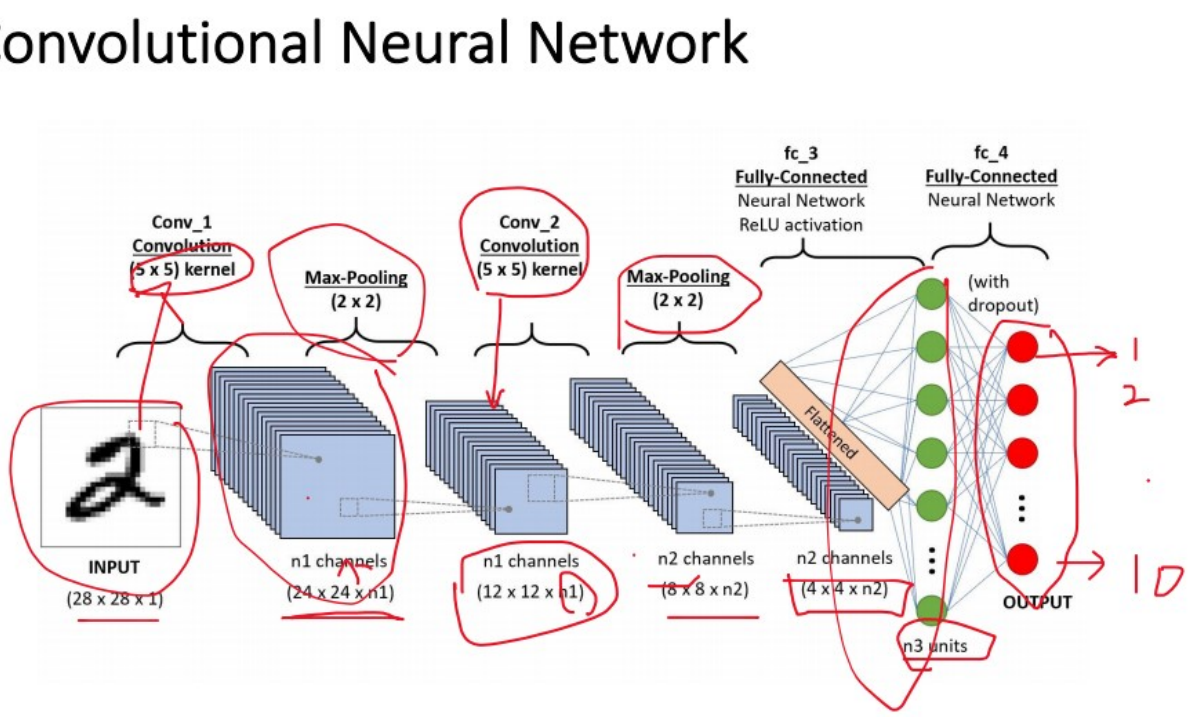
Method 1 - Chage another CNN architecture:

This architecture is defined in Lecture 4.

With kernel sizes (5 x 5), Flattened input in fc_3 and softmax at the output.

n1, n2 and n3 in this case, are 32, 64 and 128.

The CNN architecture is shown in the picture below:



(Note: if you choose to use the first FashionCNN, then don't run the following code)

```
class FashionCNN(nn.Module):
    def __init__(self):
        super(FashionCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5, 1)
        self.conv2 = nn.Conv2d(32, 64, 5, 1)
        self.dropout1 = nn.Dropout2d(0.5)
        self.fc3 = nn.Linear(4*4*64, 128)
        self.fc4 = nn.Linear(128, 10)

        # x represents our data

    def forward(self, x):
        # Pass data through conv1
        x = self.conv1(x)
        # Use the rectified-linear activation function over x
        x = F.relu(x)
        # Run max pooling over x
        x = F.max_pool2d(x, 2)
        # Pass data through conv2
        x = self.conv2(x)
        x = F.relu(x)
        # Run max pooling over x
        x = F.max_pool2d(x, 2)
        # Flatten x with start_dim=1
        x = torch.flatten(x, 1)
```

```
        x = torch.flatten(x, 1)
        # Pass data through fc3
        x = self.fc3(x)
        x = F.relu(x)
        # With dropout
        x = self.dropout1(x)
        # Pass data through fc4
        x = self.fc4(x)

        # Apply softmax to output
        output = F.log_softmax(x, dim=1)
        return output
```

Define CNN by using different architecture to construct model

Take a look of the chosen model:

```
model = FashionCNN()
model.to(device)
print(model)
```

```
FashionCNN(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=2304, out_features=600, bias=True)
  (drop): Dropout2d(p=0.25, inplace=False)
  (fc2): Linear(in_features=600, out_features=120, bias=True)
  (fc3): Linear(in_features=120, out_features=10, bias=True)
)
```

Method 2 - Change different optimizers

Firstly, define the learnin rate:

```
learningRate = 0.01
```

Secondly, choose the optimizer in torch. First one we used, is ADAM:

```
optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)
```

The second optimizer we used is SGD:

(Note, you only need to run either the code above or the following one.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate, momentum=0.9)
```

2.3 Predicting

Let us start training the network with chosen architecture and optimizer, then test it on the testing dataset

```
error = nn.CrossEntropyLoss()
numEpochs = 15
count = 0
# Lists for visualization of loss and accuracy
lossList = []
iterationList = []
accuracyList = []

# Lists for knowing classwise accuracy
predictionsList = []
labelsList = []

for epoch in range(numEpochs):
    for images, labels in trainLoader:
        # Transfering the images and labels to GPU(if available)
        images, labels = images.to(device), labels.to(device)

        train = Variable(images.view(100, 1, 28, 28))
        labels = Variable(labels)

        # Here is --> Forward pass
        outputs = model(train)
        loss = error(outputs, labels)

        # Initializing a gradient as 0 so there is no mixing of gradient among the batches
        optimizer.zero_grad()

        #Propagating the error backward
        loss.backward()

        # Optimizing the parameters (Adam Algorithm)
        optimizer.step()
```

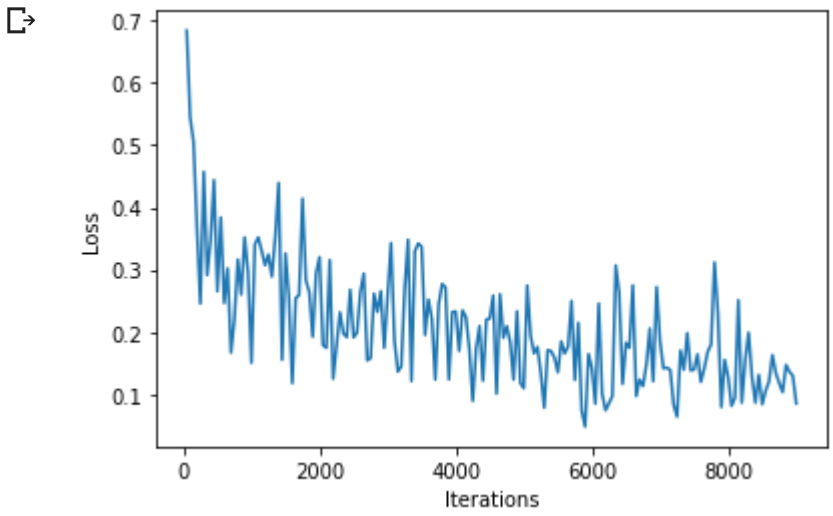
```
        optimizer.step()

        count += 1

    # Let us start testing the model:

        if not (count % 50):
            total = 0
            correct = 0

            for images, labels in testLoader:
                images, labels = images.to(device), labels.to(device)
                labelsList.append(labels)

                test = Variable(images.view(100, 1, 28, 28))

                outputs = model(test)

                predictions = torch.max(outputs, 1)[1].to(device)
                predictionsList.append(predictions)
                correct += (predictions == labels).sum()

                total += len(labels)
            accuracy = correct * 1.0 / total
            lossList.append(loss.data)
            iterationList.append(count)
            accuracyList.append(accuracy)

        if not (count % 500):
            print("Iteration: {}, Loss: {:.3f}, Accuracy: {:.2%}".format(count, loss.data, ac
```

```
Iteration: 500, Loss: 0.265, Accuracy: 86.36%
Iteration: 1000, Loss: 0.150, Accuracy: 88.40%
Iteration: 1500, Loss: 0.326, Accuracy: 89.72%
Iteration: 2000, Loss: 0.320, Accuracy: 90.50%
Iteration: 2500, Loss: 0.191, Accuracy: 90.32%
Iteration: 3000, Loss: 0.261, Accuracy: 90.05%
Iteration: 3500, Loss: 0.337, Accuracy: 90.77%
Iteration: 4000, Loss: 0.233, Accuracy: 90.64%
Iteration: 4500, Loss: 0.222, Accuracy: 90.64%
Iteration: 5000, Loss: 0.110, Accuracy: 90.47%
Iteration: 5500, Loss: 0.135, Accuracy: 91.72%
Iteration: 6000, Loss: 0.142, Accuracy: 90.32%
Iteration: 6500, Loss: 0.182, Accuracy: 90.73%
Iteration: 7000, Loss: 0.185, Accuracy: 91.04%
Iteration: 7500, Loss: 0.140, Accuracy: 91.18%
Iteration: 8000, Loss: 0.127, Accuracy: 90.62%
Iteration: 8500, Loss: 0.084, Accuracy: 91.31%
Iteration: 9000, Loss: 0.085, Accuracy: 91.31%
```

Plot the loss in iterations:

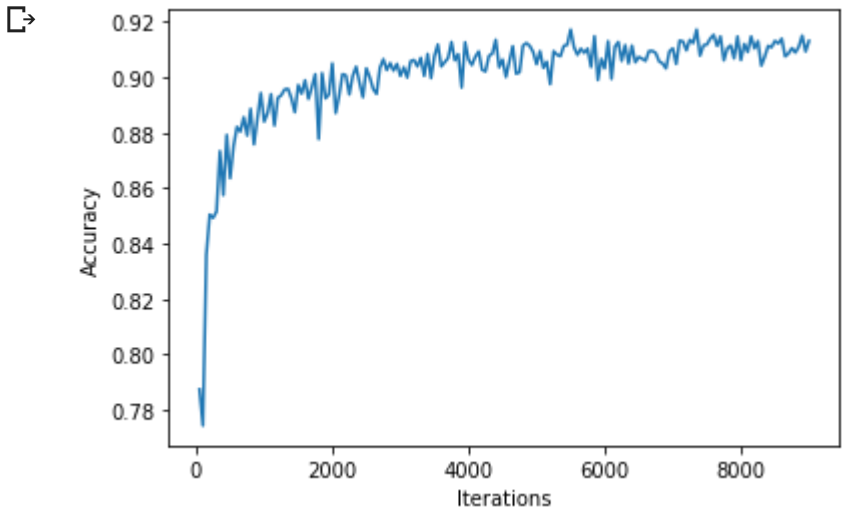
```
plt.plot(iterationList, lossList)
```

```
plt.plot(iterationList, lossList)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()
```



Plot the accuracy in iterations:

```
plt.plot(iterationList, accuracyList)
plt.xlabel("Iterations")
plt.ylabel("Accuracy")
plt.show()
```



Test

```
model.eval()
correct = 0
for images, labels in testLoader:
    with torch.no_grad(): # so that computation graph history is not stored
        images, labels = images.to(device), labels.to(device) # send tensors to GPU
```

```
        outputs = model(images)
        predictions = outputs.data.max(1)[1]
        correct += predictions.eq(labels.data).sum()

print('Test set accuracy: {:.2f}%'.format(100.0 * correct / len(testLoader.dataset)))
```

```
Test set accuracy: 91.46%
```

Auxillary: Getting the accuracy with respect to each classification in the fashionMINST Dataset

```
classCorrect = [0. for _ in range(10)]
totalCorrect = [0. for _ in range(10)]

with torch.no_grad():
    for images, labels in testLoader:
        images, labels = images.to(device), labels.to(device)
        test = Variable(images)
        outputs = model(test)
        predicted = torch.max(outputs, 1)[1]
        c = (predicted == labels).squeeze()

        for i in range(100):
            label = labels[i]
            classCorrect[label] += c[i].item()
            totalCorrect[label] += 1

for i in range(10):
    print("Accuracy of {}: {:.2f}%".format(outputLabel(i), classCorrect[i] * 100 / totalCorre
```

```
Accuracy of T-shirt/Top: 82.40%
Accuracy of Trouser: 98.50%
Accuracy of Pullover: 84.60%
Accuracy of Dress: 93.90%
Accuracy of Coat: 87.10%
Accuracy of Sandal: 98.00%
Accuracy of Shirt: 77.60%
Accuracy of Sneaker: 97.80%
Accuracy of Bag: 97.60%
Accuracy of Ankle Boot: 97.10%
```

Among 10 categories, Bag has the highest prediction accuracy, and Shirt has the lowest.

Conclusion according to Method 1

With same other conditions, the first Architecure we used, is more instable. The loss shown in the iteration has big waves.

Instead, the second Architecture defined according to the lecture, has a little bit more smooth loss curve and more stable.

Conclusion according to Method 2

For both Architecture, ADAM predicts the accuracy faster. Although, SGD is more stable in loss calculation.

Furthurmore, we found using Architecture 1 or the Architecture 2 with SGD can reach their best performance overall.

3 - Answer these questions

Question 1

Convolution makes the neural network to model translation of object in image automatically, could CNN model the different orientations of object in images? How to train a CNN to classify objects with different orientations?

Answer 1

Yes, the CNN model can be used for the different orientations of the object in images.

Firstly, we will look in brute force:

We can give the rotated images while training the model. However, we don't think it is an optimal solution. In fact, we believe this is a weak solution.

While there are different methods we can use, We will be explaining the identification of the different orientations of images using an available model named RotNet.

What is RotNet?

It is a convolutional neural network for predicting the rotation angle of an image that corrects its orientation.

To explain this model in detail:

We have two consecutive convolutional layers (with kernel size & number of the different kernels). The next layer is the max-pooling layer (which also takes the kernel size). The dropout layer sets a fraction of its input to zero.The flatten layer simply converts the 3-D input to 1-D.

Finally, we have a dropout layer & the fully connected layer.

In general, there is an activation function & the final layer uses Softmax activation.

Also, we use an ADAM optimizer used to perform the weights update.

Question 2

Random dropout is used to regularize fully connect layers and avoid overfitting of neural networks. If the dropout rate is 0.2, describing how the dropout connection performed in training and testing (backward and forward)?

Answer 2

When the dropout set to 0.2, that means in the forward propagation, 20 percent of the input data are zeroed. When backward propagation, only 80% neutrons in that layer are usable, since 20% are killed.