import torch

import torchvision

from torch.autograd import Variable

import torchvision.transforms as transforms

import torch.nn.init

import numpy as np

from sklearn.model\_selection import train\_test\_split

import json

# training set and normalizing

train\_data = torchvision.datasets.FashionMNIST("./data", train = True, download = True,

  transform = transforms.Compose([transforms.ToTensor(),

                               transforms.Normalize((0.5,),(0.5,),)]))

# test set and normalizing

test\_data = torchvision.datasets.FashionMNIST("./data", train = False,

  transform = transforms.Compose([transforms.ToTensor(),

                               transforms.Normalize((0.5,),(0.5,),)]))

#split between training set and validation set

train\_set, valid\_set = train\_test\_split(train\_data, test\_size=0.1, stratify = train\_data.targets)

#added from HW manual

train\_generator = torch.utils.data.DataLoader(train\_set, batch\_size = 50, shuffle = True) #shuffle flag is True as desired in the manual

test\_generator = torch.utils.data.DataLoader(test\_data, batch\_size = 50, shuffle = False)

valid\_generator = torch.utils.data.DataLoader(valid\_set, batch\_size = 50, shuffle = True)

#Class definitions

# example mlp classifier, taken from homework manual

class model\_mlp1(torch.nn.Module):

  def \_\_init\_\_(self):

    super(model\_mlp1, self).\_\_init\_\_()

    self.fc1 = torch.nn.Linear(784, 64)

    self.relu = torch.nn.ReLU()

    self.fc10 = torch.nn.Linear(64, 10)

  def forward(self, x):

    x = x.view(-1, 784)

    hidden = self.fc1(x)

    relu = self.relu(hidden)

    output = self.fc10(relu)

    return output

#https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html

#Sigmoid is taken from here

class model\_mlp1\_sigmoid(torch.nn.Module):

  def \_\_init\_\_(self):

    super(model\_mlp1\_sigmoid, self).\_\_init\_\_()

    self.fc1 = torch.nn.Linear(784, 64)

    self.sigmoid = torch.nn.Sigmoid()

    self.fc10 = torch.nn.Linear(64, 10)

  def forward(self, x):

    x = x.view(-1, 784)

    hidden = self.fc1(x)

    sigmoid = self.sigmoid(hidden)

    output = self.fc10(sigmoid)

    return output

class model\_mlp2(torch.nn.Module):

  def \_\_init\_\_(self):

    super(model\_mlp2, self).\_\_init\_\_()

    self.fc1 = torch.nn.Linear(784, 16)

    self.relu = torch.nn.ReLU()

    self.fc2 = torch.nn.Linear(16, 64)

    self.fc10 = torch.nn.Linear(64, 10)

  def forward(self, x):

    x = x.view(-1, 784)

    hidden = self.fc1(x)

    relu = self.relu(hidden)

    hidden2 = self.fc2(relu)

    output = self.fc10(hidden2)

    return output

class model\_mlp2\_sigmoid(torch.nn.Module):

  def \_\_init\_\_(self):

    super(model\_mlp2\_sigmoid, self).\_\_init\_\_()

    self.fc1 = torch.nn.Linear(784, 16)

    self.sigmoid = torch.nn.Sigmoid()

    self.fc2 = torch.nn.Linear(16, 64)

    self.fc10 = torch.nn.Linear(64, 10)

  def forward(self, x):

    x = x.view(-1, 784)

    hidden = self.fc1(x)

    sigmoid = self.sigmoid(hidden)

    hidden2 = self.fc2(sigmoid)

    output = self.fc10(hidden2)

    return output

class model\_cnn3\_sigmoid(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn3\_sigmoid, self).\_\_init\_\_()

        #Convolution layer definition is taken from: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 7, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 5, stride = 1, padding = 'valid')

        self.sigmoid = torch.nn.Sigmoid()

        #Pooling layer definition is taken from: https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        sigmoid1\_output = self.sigmoid(fc1\_output)

        fc2\_output = self.fc2(sigmoid1\_output)

        sigmoid2\_output = self.sigmoid(fc2\_output)

        mp1\_output = self.mp2(sigmoid2\_output)

        fc3\_output = self.fc3(mp1\_output)

        mp2\_output = self.mp2(fc3\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

class model\_cnn3(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn3, self).\_\_init\_\_()

        #Convolution layer definition is taken from: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 7, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 5, stride = 1, padding = 'valid')

        self.relu = torch.nn.ReLU()

        #Pooling layer definition is taken from: https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        relu1\_output = self.relu(fc1\_output)

        fc2\_output = self.fc2(relu1\_output)

        relu2\_output = self.relu(fc2\_output)

        mp1\_output = self.mp2(relu2\_output)

        fc3\_output = self.fc3(mp1\_output)

        mp2\_output = self.mp2(fc3\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

class model\_cnn4(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn4, self).\_\_init\_\_()

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 5, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc4 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 5, stride = 1, padding = 'valid')

        self.relu = torch.nn.ReLU()

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        relu1\_output = self.relu(fc1\_output)

        fc2\_output = self.fc2(relu1\_output)

        relu2\_output = self.relu(fc2\_output)

        fc3\_output = self.fc3(relu2\_output)

        relu3\_output = self.relu(fc3\_output)

        mp1\_output = self.mp2(relu3\_output)

        fc4\_output = self.fc4(mp1\_output)

        relu4\_output = self.relu(fc4\_output)

        mp2\_output = self.mp2(relu4\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

class model\_cnn4\_sigmoid(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn4\_sigmoid, self).\_\_init\_\_()

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 5, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc4 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 5, stride = 1, padding = 'valid')

        self.sigmoid = torch.nn.Sigmoid()

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        sigmoid1\_output = self.sigmoid(fc1\_output)

        fc2\_output = self.fc2(sigmoid1\_output)

        sigmoid2\_output = self.sigmoid(fc2\_output)

        fc3\_output = self.fc3(sigmoid2\_output)

        sigmoid3\_output = self.sigmoid(fc3\_output)

        mp1\_output = self.mp2(sigmoid3\_output)

        fc4\_output = self.fc4(mp1\_output)

        sigmoid4\_output = self.sigmoid(fc4\_output)

        mp2\_output = self.mp2(sigmoid4\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

class model\_cnn5(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn5, self).\_\_init\_\_()

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc4 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc5 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc6 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.relu = torch.nn.ReLU()

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        relu1\_output = self.relu(fc1\_output)

        fc2\_output = self.fc2(relu1\_output)

        relu2\_output = self.relu(fc2\_output)

        fc3\_output = self.fc3(relu2\_output)

        relu3\_output = self.relu(fc3\_output)

        fc4\_output = self.fc4(relu3\_output)

        relu4\_output = self.relu(fc4\_output)

        mp1\_output = self.mp2(relu4\_output)

        fc5\_output = self.fc5(mp1\_output)

        relu5\_output = self.relu(fc5\_output)

        fc6\_output = self.fc6(relu5\_output)

        relu6\_output = self.relu(fc6\_output)

        mp2\_output = self.mp2(relu6\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

class model\_cnn5\_sigmoid(torch.nn.Module):

    def \_\_init\_\_(self):

        super(model\_cnn5\_sigmoid, self).\_\_init\_\_()

        self.fc1 = torch.nn.Conv2d(in\_channels = 1, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc2 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc3 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc4 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 8, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc5 = torch.nn.Conv2d(in\_channels = 8, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.fc6 = torch.nn.Conv2d(in\_channels = 16, out\_channels = 16, kernel\_size = 3, stride = 1, padding = 'valid')

        self.sigmoid = torch.nn.Sigmoid()

        self.mp2 = torch.nn.MaxPool2d(kernel\_size = 2, stride=2)

        self.fc10 = torch.nn.Linear(144,10)

    def forward(self, x):

        fc1\_output = self.fc1(x)

        sigmoid1\_output = self.sigmoid(fc1\_output)

        fc2\_output = self.fc2(sigmoid1\_output)

        sigmoid2\_output = self.sigmoid(fc2\_output)

        fc3\_output = self.fc3(sigmoid2\_output)

        sigmoid3\_output = self.sigmoid(fc3\_output)

        fc4\_output = self.fc4(sigmoid3\_output)

        sigmoid4\_output = self.sigmoid(fc4\_output)

        mp1\_output = self.mp2(sigmoid4\_output)

        fc5\_output = self.fc5(mp1\_output)

        sigmoid5\_output = self.sigmoid(fc5\_output)

        fc6\_output = self.fc6(sigmoid5\_output)

        sigmoid6\_output = self.sigmoid(fc6\_output)

        mp2\_output = self.mp2(sigmoid6\_output).view(50,144)

        fc10\_output = self.fc10(mp2\_output)

        return fc10\_output

#Class definitions

#https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/

device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

#Change of device is taken from here to compute trainings faster

#initializing the models

mlp1 = model\_mlp1().to(device)

mlp2 = model\_mlp2().to(device)

cnn3 = model\_cnn3().to(device)

cnn4 = model\_cnn4().to(device)

cnn4\_lr01 = model\_cnn4().to(device)

cnn4\_lr001 = model\_cnn4().to(device)

cnn4\_lr0001 = model\_cnn4().to(device)

cnn5 = model\_cnn5().to(device)

mlp1\_sigmoid = model\_mlp1\_sigmoid().to(device)

mlp2\_sigmoid = model\_mlp2\_sigmoid().to(device)

cnn3\_sigmoid = model\_cnn3\_sigmoid().to(device)

cnn4\_sigmoid = model\_cnn4\_sigmoid().to(device)

cnn5\_sigmoid = model\_cnn5\_sigmoid().to(device)

#initializing the models

#Training function for the q2

def train(model,model\_name):

    criterion = torch.nn.CrossEntropyLoss()

    training\_loss\_overall = []

    training\_accuracy\_overall = []

    validation\_accuracy\_overall = []

    max\_weight = 0

    max\_accuracy = 0

    print('TrainingStarts')

    for steps in range (0,1):

      print('step++')

      optimizer = torch.optim.Adam(model.parameters())

      model\_training\_loss = []

      model\_training\_accuracy = []

      validation\_accuracy\_list = []

      epochs = 15

#https://medium.com/@aaysbt/fashion-mnist-data-training-using-pytorch-7f6ad71e96f4

#The website above is used for the training, validation and test steps

#Some parts of it are directly taken.

      for epoch in range (0,epochs):

        print('epoch++')

        model.train()

        for train\_batch\_num, (images, labels) in enumerate(train\_generator):

          x = Variable(images).to(device)

          y = Variable(labels).to(device)

          optimizer.zero\_grad()

          outputs = model(x)

          loss = criterion(outputs, y)

          loss.backward()

          optimizer.step()

          #Evaluating the model in each 10 steps

          if train\_batch\_num % 10 == 0:

            model.eval()

            prediction = outputs.data.max(dim=1)[1]

            model\_training\_accuracy.append((((prediction.data == y.data).float().mean()).item())\*100)

            model\_training\_loss.append(loss.item())

            correct = 0

            sample = 0

            for validation\_batch\_num, (images, labels) in enumerate(valid\_generator):

              x=Variable(images).to(device)

              y=Variable(labels).to(device)

              score = model(x)

              \_,prediction = score.max(1)

              correct += (prediction.data == y.data).sum()

              sample += prediction.size(0)

            validation\_accuracy = float(correct) / float(sample) \* 100

            validation\_accuracy\_list.append(validation\_accuracy)

            model.train()

      #Appending the data with relative lists.

      training\_loss\_overall.append(model\_training\_loss)

      training\_accuracy\_overall.append(model\_training\_accuracy)

      validation\_accuracy\_overall.append(validation\_accuracy\_list)

      model.eval()

      true = 0

      sample = 0

      for test\_batch\_num, (images, labels) in enumerate(test\_generator):

        x=Variable(images).to(device)

        y=Variable(labels).to(device)

        score = model(x)

        \_,prediction = score.max(1)

        true += (prediction.data == y.data).sum()

        sample += prediction.size(0)

        accuracy = float(true) / float(sample) \* 100

        if accuracy >= max\_accuracy:

          max\_accuracy = accuracy

          #Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!

          #I took the error above, hence, I switched between cpu and device when I get the error in the following of the code.

          model.to('cpu')

          #https://discuss.pytorch.org/t/access-weights-of-a-specific-module-in-nn-sequential/3627 taken from here as the first layer of the models are fc1

          max\_weight = model.fc1.weight.data.numpy()

          model.to(device)

      #Create dictionary (Created by observing utils.py)

      dictionary = {"name": model\_name,

                    "loss\_curve":  np.mean(training\_loss\_overall, axis=0).tolist(),

                    "train\_acc\_curve": np.mean(training\_accuracy\_overall, axis=0).tolist(),

                  "val\_acc\_curve": np.mean(validation\_accuracy\_overall, axis=0).tolist(),

                  "test\_acc": max\_accuracy,

                  "weights":max\_weight.tolist()}

    #Write on json file

    #https://pythonexamples.org/python-write-json-to-file/

      with open('C:/Users/Can/Desktop/EE449\_HW1'+'part2\_'+  model\_name +'.json', 'w') as json\_file:

          json.dump(dictionary, json\_file)

          json\_file.close()

    print('TrainingEnds')

    return

#Training function for the q2

train(mlp1,'mlp1')

train(mlp2,'mlp2')

train(cnn3,'cnn3')

train(cnn4,'cnn4')

train(cnn5,'cnn5')

#Training function for the q3

def train\_q3(model,model\_name, model\_type):

    criterion = torch.nn.CrossEntropyLoss()

    #Definitions are taken from homework manual

    relu\_loss = []

    sigmoid\_loss = []

    relu\_grad = []

    sigmoid\_grad = []

    print('TrainingStarts')

    for steps in range (0,1):

      print('step++')

      lr = 0.01

      optimizer = torch.optim.SGD(model.parameters(), lr, momentum = 0)

      epochs = 15

      for epoch in range (0,epochs):

        print('epoch++')

        model.train()

        for train\_batch\_num, (images, labels) in enumerate(train\_generator):

          x = Variable(images).to(device)

          y = Variable(labels).to(device)

          outputs = model(x)

          loss = criterion(outputs, y)

          model.to('cpu')

          first\_layer\_first\_weight = model.fc1.weight.data.numpy()

          model.to(device)

          optimizer.zero\_grad()

          loss.backward()

          optimizer.step()

          if train\_batch\_num % 10 == 0:

            model.eval()

            model.to('cpu')

            first\_layer\_last\_weight = model.fc1.weight.data.numpy()

            model.to(device)

            #https://pytorch.org/tutorials/beginner/blitz/neural\_networks\_tutorial.html

            #grad to append calculation is taken from this site.

            weight\_difference = first\_layer\_last\_weight - first\_layer\_first\_weight

            grad = weight\_difference / lr

            grad\_to\_append = np.linalg.norm(grad).tolist()

            #According to the parameter of the function

            #Data is added to the related list

            if model\_type == 'sigmoid':

                sigmoid\_loss.append(loss.item())

                sigmoid\_grad.append(grad\_to\_append)

            if model\_type == 'relu':

                relu\_loss.append(loss.item())

                relu\_grad.append(grad\_to\_append)

            model.train()

      #Create dictionary (Created by observing utils.py)

      dictionary = {"name": model\_name,

                    "relu\_loss\_curve":  relu\_loss,

                    "sigmoid\_loss\_curve": sigmoid\_loss,

                    "relu\_grad\_curve": relu\_grad,

                    "sigmoid\_grad\_curve": sigmoid\_grad}

    #Write on json file

    #https://pythonexamples.org/python-write-json-to-file/

      with open('C:/Users/Can/Desktop/EE449\_HW1'+'part3\_'+  model\_name +'.json', 'w') as json\_file:

          json.dump(dictionary, json\_file)

          json\_file.close()

    print('TrainingEnds')

    return

#json files are combined manually

train\_q3(mlp1,'mlp1','relu')

train\_q3(mlp1\_sigmoid,'mlp1\_sigmoid','sigmoid')

train\_q3(mlp2,'mlp2','relu')

train\_q3(mlp2\_sigmoid,'mlp2\_sigmoid','sigmoid')

train\_q3(cnn3,'cnn3','relu')

train\_q3(cnn3\_sigmoid,'cnn3\_sigmoid','sigmoid')

train\_q3(cnn4,'cnn4','relu')

train\_q3(cnn4\_sigmoid,'cnn4\_sigmoid','sigmoid')

train\_q3(cnn5,'cnn5','relu')

train\_q3(cnn5\_sigmoid,'cnn5\_sigmoid','sigmoid')

#json files are combined manually

#Training function for the q3

#Training function for the q4\_part1

def train\_q4\_part1(model,model\_name,inp\_lr):

    loss\_curve\_1 = []

    loss\_curve\_01 = []

    loss\_curve\_001 = []

    val\_acc\_curve\_1 = []

    val\_acc\_curve\_01 = []

    val\_acc\_curve\_001 = []

    #I tried without inp\_lr parameter. However, I could not reset the model.

    #Hence, I decided to took it as a parameter

    learning\_rates = [inp\_lr]

    #For loop is because of the first implementation attempt

    for lr in learning\_rates:

        criterion = torch.nn.CrossEntropyLoss()

        #Data observation

        print('TrainingStarts\n')

        print(lr)

        #Data observation

        model\_training\_loss = []

        model\_training\_accuracy = []

        validation\_accuracy\_list = []

        #Defined epochs in order to test the function first.

        #e.g I gave epochs = 3 and observed the behavior.

        epochs = 20

        for epoch in range (0,epochs):

          print('epoch++')

          model.train()

          #Adjusting the optimizer with respect to lr

          optimizer = torch.optim.SGD(model.parameters(), lr, momentum = 0)

          print(lr)

          for train\_batch\_num, (images, labels) in enumerate(train\_generator):

            x = Variable(images).to(device)

            y = Variable(labels).to(device)

            optimizer.zero\_grad()

            outputs = model(x)

            loss = criterion(outputs, y)

            loss.backward()

            optimizer.step()

            if train\_batch\_num % 10 == 0:

              model.eval()

              prediction = outputs.data.max(dim=1)[1]

              model\_training\_accuracy.append((((prediction.data == y.data).float().mean()).item())\*100)

              model\_training\_loss.append(loss.item())

              correct = 0

              sample = 0

              for validation\_batch\_num, (images, labels) in enumerate(valid\_generator):

                x=Variable(images).to(device)

                y=Variable(labels).to(device)

                score = model(x)

                \_,prediction = score.max(1)

                correct += (prediction.data == y.data).sum()

                sample += prediction.size(0)

              validation\_accuracy = float(correct) / float(sample) \* 100

              validation\_accuracy\_list.append(validation\_accuracy)

              model.train()

          #Appending data with respect to learning rate of the optimizer

          if lr == 0.1:

                loss\_curve\_1 = model\_training\_loss

                val\_acc\_curve\_1 = validation\_accuracy\_list

          if lr == 0.01:

                loss\_curve\_01 = model\_training\_loss

                val\_acc\_curve\_01 = validation\_accuracy\_list

          if lr == 0.001:

                loss\_curve\_001 = model\_training\_loss

                val\_acc\_curve\_001 = validation\_accuracy\_list

    #Create dictionary (Created by observing utils.py)

    dictionary = {"name": model\_name,

                  "loss\_curve\_1":  loss\_curve\_1,

                  "loss\_curve\_01":  loss\_curve\_01,

                "loss\_curve\_001":  loss\_curve\_001,

                "val\_acc\_curve\_1": val\_acc\_curve\_1,

                "val\_acc\_curve\_01": val\_acc\_curve\_01,

                "val\_acc\_curve\_001": val\_acc\_curve\_001}

    #Write on json file

    #https://pythonexamples.org/python-write-json-to-file/

    with open('C:/Users/Can/Desktop/EE449\_HW1'+'part4\_2LR\_'+  model\_name + 'lr = {}'.format(lr) + '.json', 'w') as json\_file:

        json.dump(dictionary, json\_file)

        json\_file.close()

    print('TrainingEnds')

    return

train\_q4\_part1(cnn4\_lr01,'cnn4\_lr01',0.1)

train\_q4\_part1(cnn4\_lr001,'cnn4\_lr001',0.01)

train\_q4\_part1(cnn4\_lr0001,'cnn4\_lr0001',0.001)

#json files are combined manually

#Training function for the q4\_part1

#Training function for the q4\_part2

def train\_q4\_part2(model,model\_name,inp\_lr):

    #Necessary list

    loss\_curve\_1 = []

    loss\_curve\_01 = []

    loss\_curve\_001 = []

    val\_acc\_curve\_1 = []

    val\_acc\_curve\_01 = []

    val\_acc\_curve\_001 = []

    learning\_rates = [inp\_lr]

    for lr in learning\_rates:

        criterion = torch.nn.CrossEntropyLoss()

        print('TrainingStarts\n')

        print(lr)

        model\_training\_loss = []

        model\_training\_accuracy = []

        validation\_accuracy\_list = []

        epochs = 30

        for epoch in range (0,epochs):

          #Updating the learning rate with the epoch that is observed in the previous stages

          if epoch == 7:

            lr = 0.01

          #Updating the learning rate with the epoch that is observed in the previous stages

          if epoch == 14:

            lr == 0.001

          print('epoch++')

          model.train()

          #Updating optimizer with parameters

          optimizer = torch.optim.SGD(model.parameters(), lr, momentum = 0)

          print(lr)

          for train\_batch\_num, (images, labels) in enumerate(train\_generator):

            x = Variable(images).to(device)

            y = Variable(labels).to(device)

            optimizer.zero\_grad()

            outputs = model(x)

            loss = criterion(outputs, y)

            loss.backward()

            optimizer.step()

            if train\_batch\_num % 10 == 0:

              model.eval()

              prediction = outputs.data.max(dim=1)[1]

              model\_training\_accuracy.append((((prediction.data == y.data).float().mean()).item())\*100)

              model\_training\_loss.append(loss.item())

              correct = 0

              sample = 0

              for validation\_batch\_num, (images, labels) in enumerate(valid\_generator):

                x=Variable(images).to(device)

                y=Variable(labels).to(device)

                score = model(x)

                \_,prediction = score.max(1)

                correct += (prediction.data == y.data).sum()

                sample += prediction.size(0)

              validation\_accuracy = float(correct) / float(sample) \* 100

              validation\_accuracy\_list.append(validation\_accuracy)

              model.train()

          if lr == 0.1:

                loss\_curve\_1 = model\_training\_loss

                val\_acc\_curve\_1 = validation\_accuracy\_list

          if lr == 0.01:

                loss\_curve\_01 = model\_training\_loss

                val\_acc\_curve\_01 = validation\_accuracy\_list

          if lr == 0.001:

                loss\_curve\_001 = model\_training\_loss

                val\_acc\_curve\_001 = validation\_accuracy\_list

    #Create dictionary (Created by observing utils.py)

    dictionary = {"name": model\_name,

                  "loss\_curve\_1":  loss\_curve\_1,

                  "loss\_curve\_01":  loss\_curve\_01,

                "loss\_curve\_001":  loss\_curve\_001,

                "val\_acc\_curve\_1": val\_acc\_curve\_1,

                "val\_acc\_curve\_01": val\_acc\_curve\_01,

                "val\_acc\_curve\_001": val\_acc\_curve\_001}

    #Write on json file

    #https://pythonexamples.org/python-write-json-to-file/

    with open('C:/Users/Can/Desktop/EE449\_HW1'+'part4\_2LR\_'+  model\_name + 'lr = {}'.format(lr) + '.json', 'w') as json\_file:

        json.dump(dictionary, json\_file)

        json\_file.close()

    print('TrainingEnds')

    return

train\_q4\_part2(cnn4\_lr01,'cnn4',0.1)

#Training function for the q4\_part2