MIDDLE EAST TECHNICAL UNIVERSITY

EE449 Computational Intelligence

Homework 1

Can Aydın

2304053

Section-1

Table of Contents

Question 1)

1-)

An ANNs classifier trained with cross-entropy loss approximates the given input size into the pre-defined classses. In this homework, our input size was 784 and number of classes was 10. Hence, this function approximates a set with size 784 into another set with size 10. In addition, cross-entropy loss is defined as the following equation:

- H(P, Q) = – sum x in X P(x) * log(Q(x))

where, x is between 1 and the number of classes, Q(x) is the observation probability of a class and P(x) is the probability of the correct observation of a class. The reason behind that is, the loss equation should detect the correct classification. When we predict the class correctly, our loss will be log(1) which is equal to zero. However, it will be equal to log(0) which equals to (-infinity) for a wrong case. We can detect the correctness with this two values of the loss.

2-) Gradient can be computed from the equation in figure.1:

$$\Delta \mathbf{w}(t) = -\eta \nabla e(\mathbf{w}(t))$$

Figure.1: Gradient calculation from lecture notes where n is learning rate.

3-)

3.1) Batch size is the total number of the examples that are given to the model in one iteration.

Epochs are the total number of iterations that is computed to train the model with all the training data.

3.2) N/B

3.3) (N*E)/B

4) 4.1) Parameter number is the total number for interpretable feature for a given layer. Hence, in order to calculate the overall parameter number, we should calculate each layers' parameter number and sum them. In order to compute that, we should calculate the connections (weights) between consecutive layers' neurons. In addition, we should consider the connections between bias neurons and layer neurons. Overall, we can calculate the total number of parameters in the following formula:

$$i \times h1 + \sum_{k=1}^{n-1} (h_k \times h_{k+1}) + h_n \times o + \sum_{k=1}^{n} h_k + o$$

where i is the input layer, n is the number of hidden layers and o is the number of output layer. Note that, we should add the connection number of bias neurons to this formula if we have them in the neural network.

4.2) We can calculate the number of parameters that CNN has with the formula similar to the above.

$$Total\ Number\ of\ Parameters = \sum_{i=1}^{k-1} (H_i * W_i * C_i + 1) * C_{i+1}$$

Question 2)

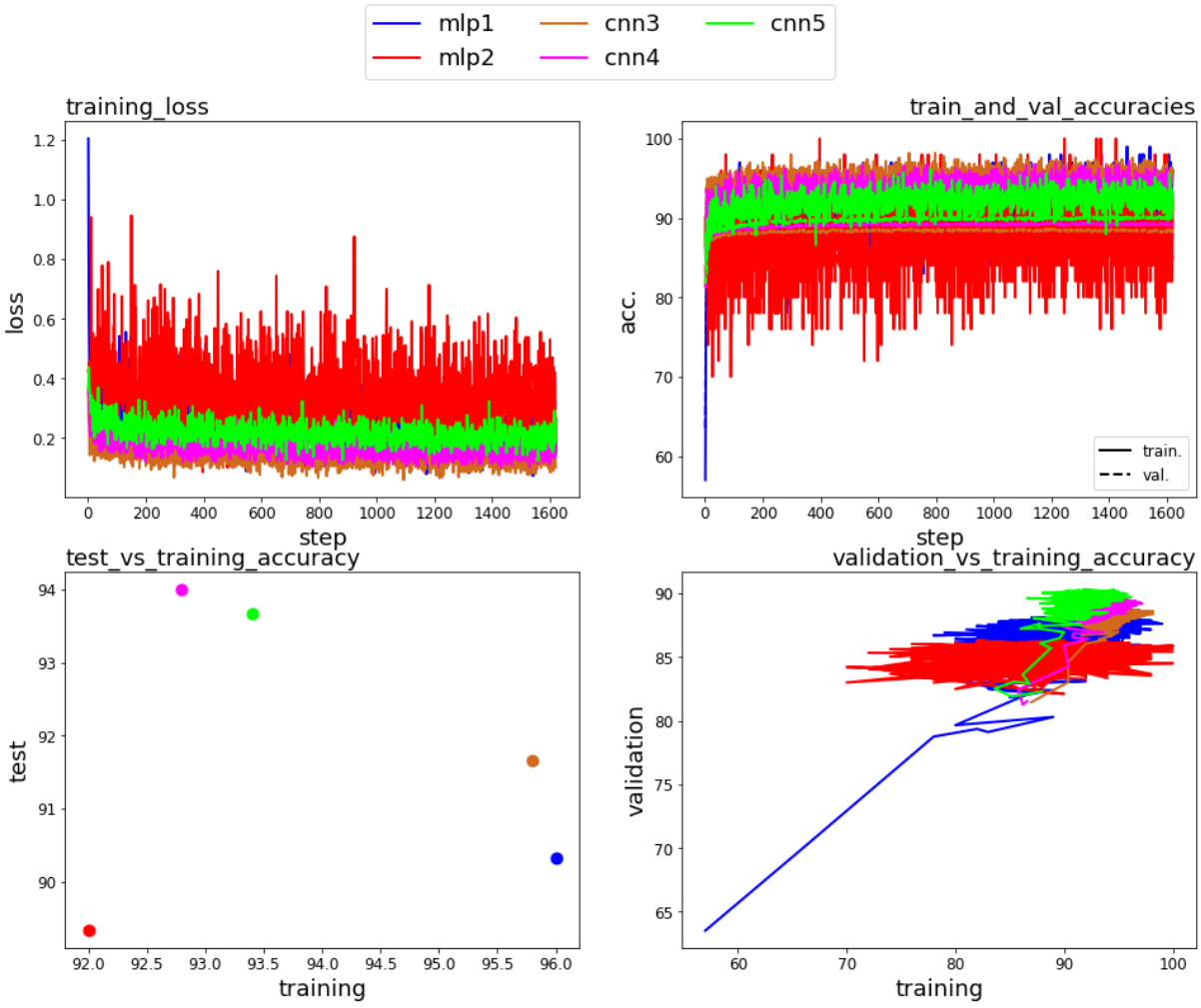

Figure.2: Plots of training, test and validation results of the question 2.

Figure.3: Visualization of weights for mlp1 model.

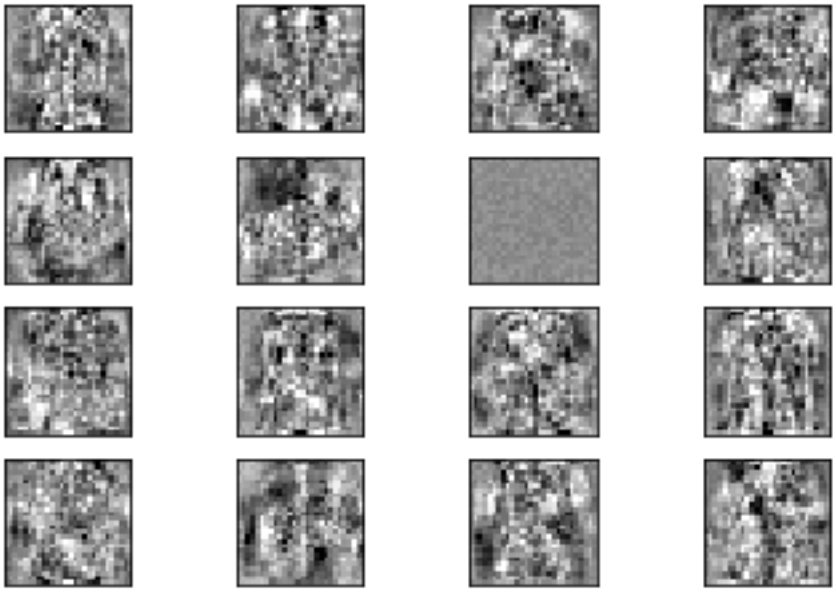

Figure.4: Visualization of weights for mlp2 model.

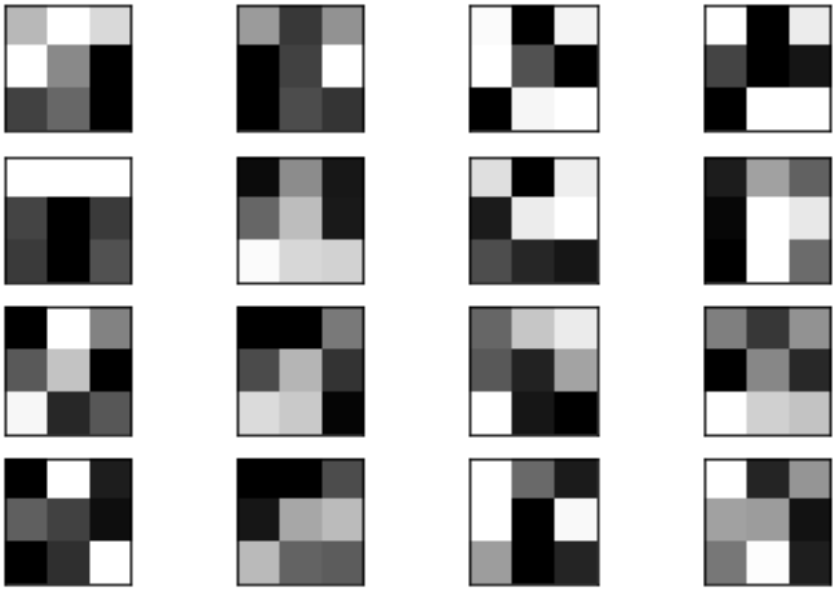

Figure.5: Visualization of weights for cnn3 model.

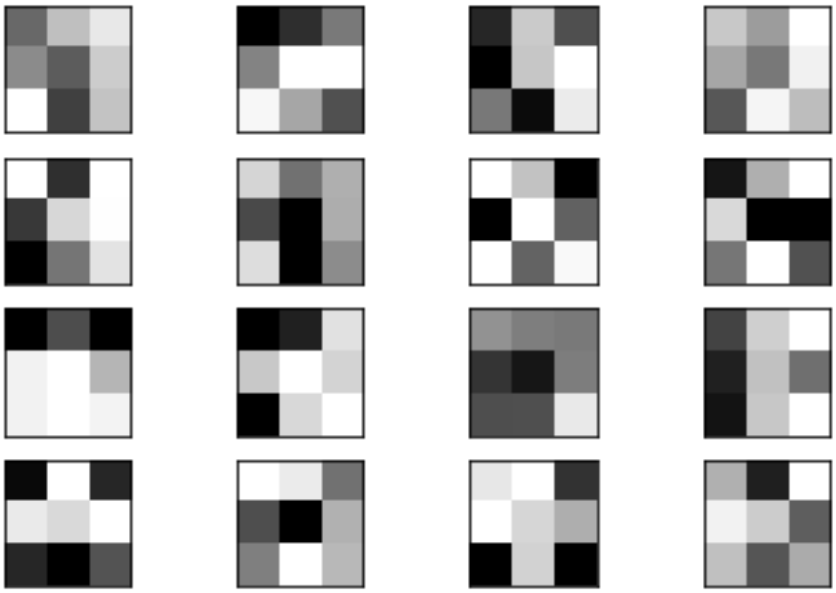

Figure.6: Visualization of weights for cnn4 model.

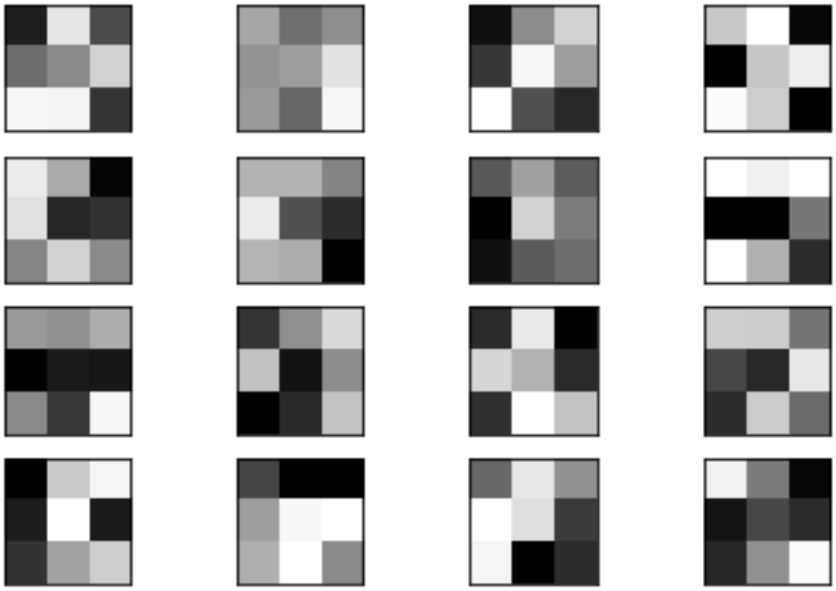

Figure.7: Visualization of weights for cnn5 model.

Discussion

1) Generalization performance is a measurement of classifier's capability to learn from the data set while training and successfully determine the classes inputs that is given to the model for the first time. In other words, it is a measurement of how successful the model is while generalizing the training data set.

2) Generalization performance can be inspected from validation and test accuracy plots. The reason behind that, as mentioned in previous question, generalization is about the how the training reflects to the future inputs. Therefore, it should be inspected after training. Thus, we should observe validation and test accuracy plots.

3) CNN4 model has the most test accuracy and CNN5 model has the most validation accuracy. In addition, MLP2 model has the lowest validation and test accuracy. Hence, the best performances belong to CNN4 and CNN5 and the worst performance belongs to MLP2 model with respect to generalization performance.(Observe figure.2)

4) According to the Grosse, the generalization performance is high, then algorithm is successful. However, if the performance is low, model memorizes the training data. Hence, it cannot interpret the incoming data. The more parameters, the more model memorizes the data. In this case, CNN4 and CNN5 has the lowest parameters which shows the best performances.

5) Depth of the architecture is proportional with the performance. In this homework, CNN5 and CNN4 has the highest depth. As can be seen from the figure 2, the best performances, again, belong to CNN4 and CNN5.

6) As can be seen from figure 3 to figure 7, weight visualizations of the MLP models are hard to interpret. In addition, visualizations of CNN models are not interpretable.

7) During visualization, we only considered one layer. If we considered all the layers, classes of units might be determined. However, it is not possible from one layer.

8) Weights of MLP architecture are more interpretable.

9) When we consider CNN models, the number of parameters is inversely proportional with the index number of model (CNN'5', e.g.). In addition, depth of the model is proportional with the index number. Hence, we expect to observe the best performance on CNN5 model. However, CNN4 model has better train results. On the other hand, we expect to have better performance on MLP2 model compared to MLP1. However, MLP 1 has a better performance. Overall, CNN models have better perforamance than MLP models. The reason behind that might be the higher layer number of CNN models.

10) CNN5 model would be chosen for the classification task, since, in theory, it should have the best performance. In addition, when we observe figure 2, theory coincides with the practical usage. Hence, CNN5 is the best choice.

Question 3)

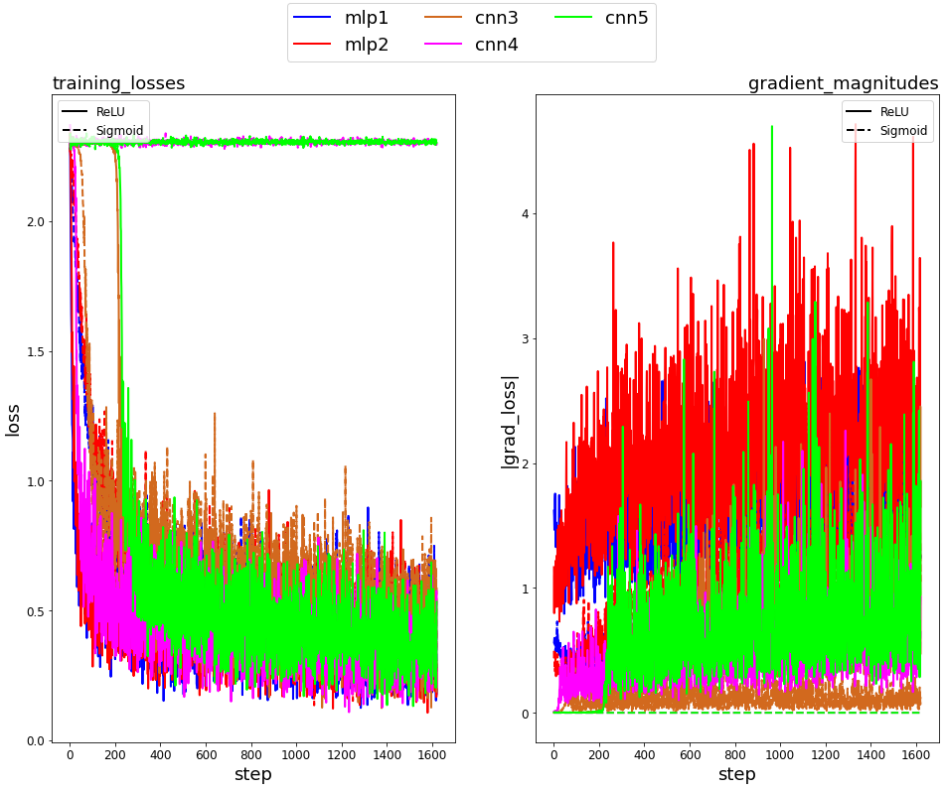

Figure.8: Training losses and gradient magnitudes graph for two different activation function.

Discussion:

1) As can be seen from figure 8, models with ReLU activation function generates higher gradients in magnitude. In addition, MLP models have more gradients when the same activation function is used. When we compare the depth of the models, CNN5 has the most depth. As can be seen from figure 8, right-hand side plot, it also has the lowest gradient. Hence, we can conclude that depth of the model is inversely proportional with the gradient magnitude.

2) As mentioned before, it is important for the model to be able to generalize the training data which means "learning" from the data. Moreover, the more depth of the model, the better model can interpret the incoming data. The reason behind that, depth helps the model to calculate the best weights and reach the best performance. Therefore, CNN models have better performance.

When we compare the activation functions' behavior, sigmoid activation function generates output between 0 and 1. Therefore, it has lower gradient compared to the ReLU activation function as its' output varies between 0 and ∞.

3) If we do not scale the inputs to the range [-1.0, 1.0], it will last longer for model to reach a better performance. Hence, gradients magnitudes would be higher compared to the figure 8, right-hand side plot.

Question 4)

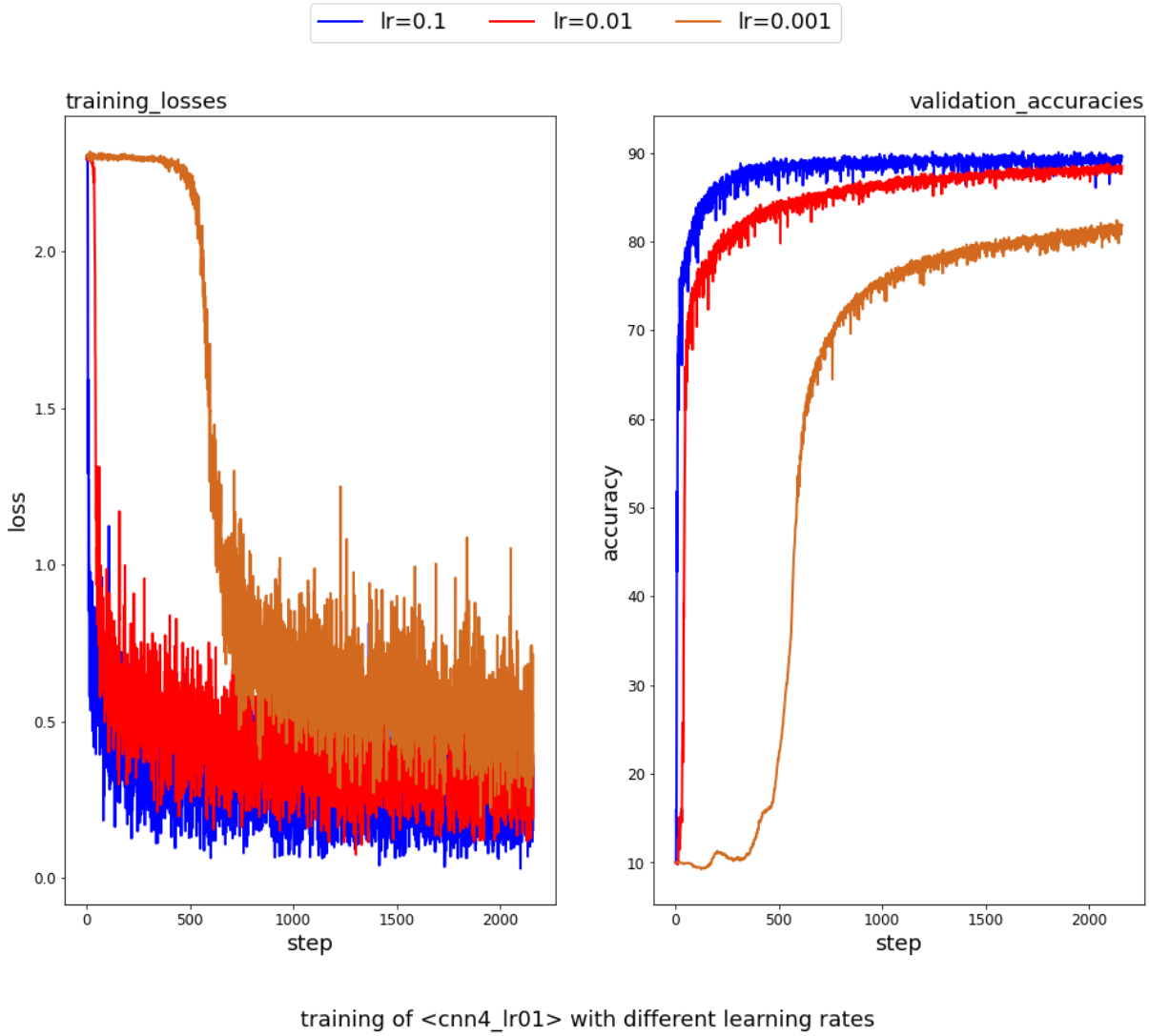

Figure.9: Training of cnn4 model with different learning rates.

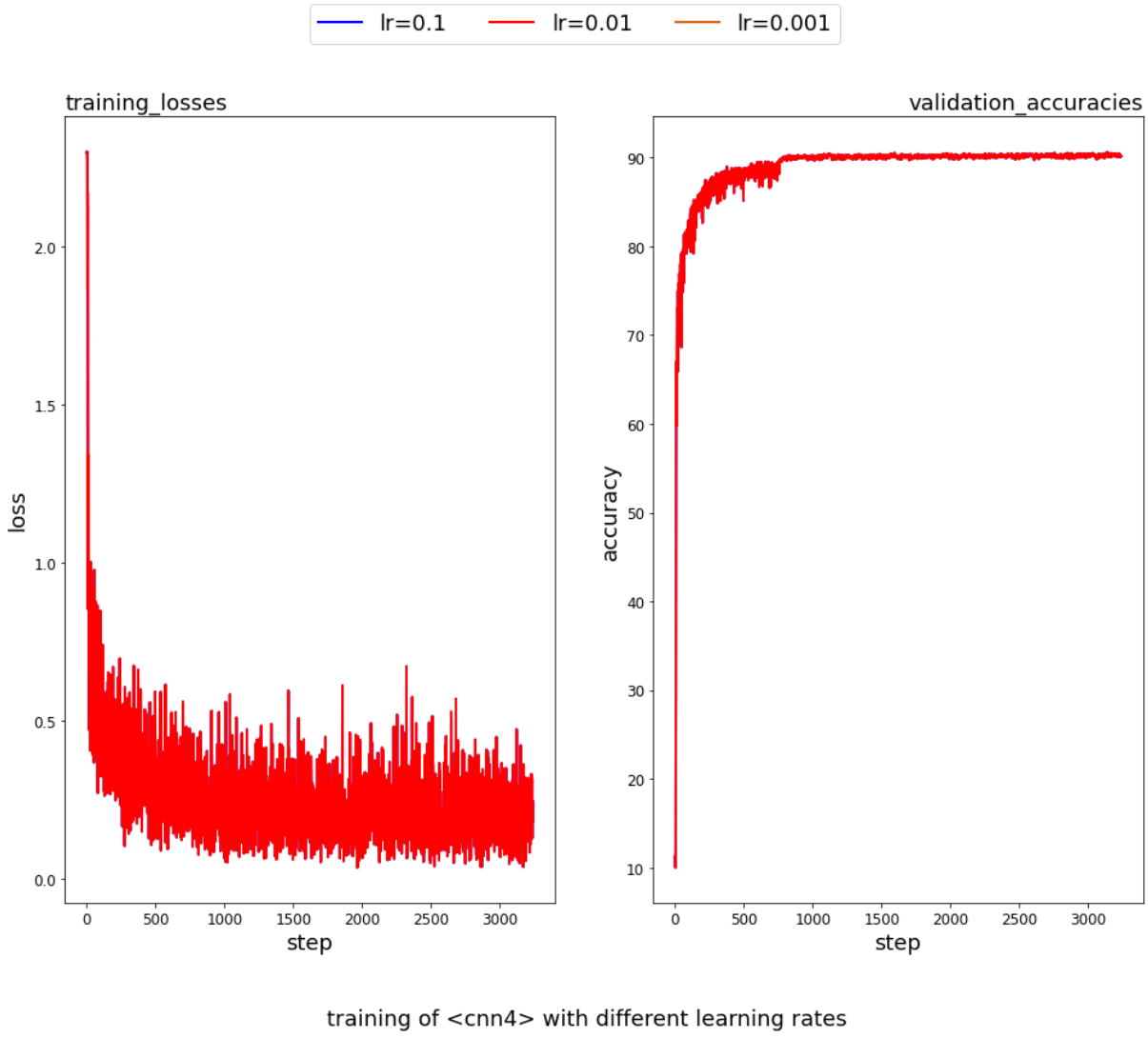

Figure.10: Training of cnn4 model with changing learning rates. (lr = 0.1, 0.01)

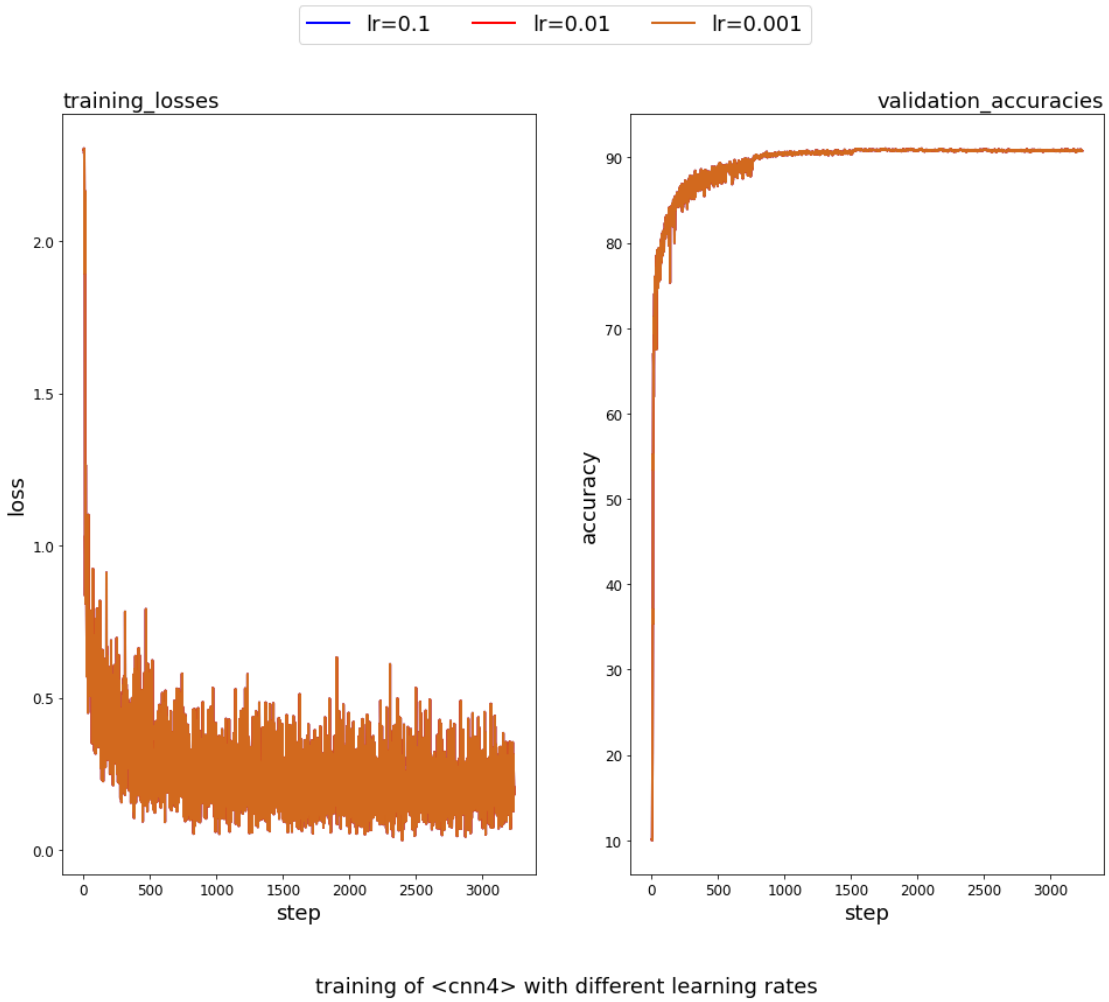

Figure.11: Training of cnn4 model with changing learning rates. (lr = 0.1, 0.01, 0.001)

Discussion:

1) As can be understood from the name of "learning rate", it is proportional with the convergence speed.

2) When we observe figure 11, right-hand side plot, when learning rate is 0.1, we have more oscillation. Moreover, when learning rate is 0.01, we have less oscillation at the point the curve converges to a certain accuracy. However, the less oscillation we have at convergence point, the more it lasts to reach to that point. As in Control Theory, we should sacrifice speed to reach a more stable point. Therefore, we should analyze the plots and find an optimum point for both speed and stability of the convergence point.

3) Scheduled learning rate method works in this case. Since, we change the learning rate at the points where accuracy reaches a certain point, we eliminate the disadvantages. To be more precise, we use the speed of the learning rate 0.1 at the beginning and we use the stability of the learning rate 0.001 at the end. Overall, we create a controller, in a sense for learning rate to maximize the model's performance.

4) As can be seen from the plots, convergence performance is better when we use scheduled (adaptive) learning rate method. However, accuracy values are nearly the same when we compare scheduled learning rate method with Adam.

Code of the Homework

Code of Common Parts that are used by the train functions

```
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
```

Code of Question 2

```
#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')
```

Code of Question 3

```
#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 and lecture notes.
        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
```

Code of Question 4 for Different Learning Rates

```
#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
```

Code of Question 4 for Changing Learning Rates

```
#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
     #In order to obtain the previous step, training only with lr = 0.1 and 0.01, just delete
     #the next if statement. Other than that, the code is the same. Hence, I did not create
     #two different functions for this step. I did not add the code for only lr=0.1 and 0.01
     #to prevent a mess of codes.
     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
```

References

Arun-purakkatt. (2020, August 19). *Deep_Learning_Pytorch/fashion_mnist_feedforward_nn.ipynb at master · Arun-Purakkatt/deep_learning_pytorch*. GitHub. Retrieved April 25, 2022, from https://github.com/Arun-purakkatt/Deep_Learning_Pytorch/blob/master/Fashion_MNIST_feedforward_NN.ipynb

Brownlee, J. (2020, December 22). *A gentle introduction to cross-entropy for Machine Learning*. Machine Learning Mastery. Retrieved April 25, 2022, from https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Convolutional Neural Network Pytorch: CNN using pytorch. Analytics Vidhya. (2020, May 10). Retrieved April 25, 2022, from https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/

Google. (n.d.). *Google colaboratory*. Google Colab. Retrieved April 25, 2022, from https://colab.research.google.com/drive/1nkAZaX5QQhnO3XN3FE27jW3JKk0tcPd_?usp=sharing#scrollTo=--9rCNYRynEu

How do you calculate the number of parameters of ... - quora. (n.d.). Retrieved April 25, 2022, from https://www.quora.com/How-do-you-calculate-the-number-of-parameters-of-an-MLP-neural-network

Lecture 9: Generalization - Department of Computer Science ... (n.d.). Retrieved April 25, 2022, from https://www.cs.toronto.edu/~lczhang/321/notes/notes09.pdf

Vasudev, R. (2020, May 25). *Understanding and calculating the number of parameters in Convolution Neural Networks (cnns)*. Medium. Retrieved April 25, 2022, from https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d