EEE-443 Neural Networks Project-2

Name: Ayberk Yarkın

Surname: Yıldız

Id: 21803386

Date: November 29, 2021

Ayberk Yarkın Yıldız
21803386
EEE443 Project-2

Question 1

In this question, it is asked to implement a neural network that classifies the images either to cat or car using Stochastic Gradient Descent on mini-batches. Mean squared and mean classification errors are used. Mean squared error (MSE) is defined below:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y^i - \hat{y}^i)^2$$

where there is a total of N samples to be calculated.

Mean classification error (MCE) is determined with the percentage of correctly classified images.

a) In part a, it is asked to design a multi-layer neural network with a single hidden and output layer using backpropagation algorithm. It is assumed a hyperbolic tangent activation function for all neurons. The tanh activation function can be derived below:

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Although activation function can be derived as the formula above, I used tanh function of numpy in my code. The derivative of the activation function that will be used in backpropagation is shown below:

$$\frac{\partial \tanh(x)}{\partial x} = 1 - tanh^2(x) = 1 - y^2$$

The update formula of the weights is shown below:

$$\Delta W = \eta * \delta * X^T$$

where the delta $\delta = \frac{\partial \tanh(x)}{\partial x} \odot \frac{\partial E}{\partial Y}$ is calculated individually for hidden and output layers, where the error is calculated as:

$$E = \frac{1}{2}(Y - \hat{Y})$$

and X is the input for the hidden or the output layer.

My optimized values for the network are shown below:

Ayberk Yarkın Yıldız
21803386
EEE443 Project-2

Parameter	Value
Hidden Neuron Number	30
Output Neuron Number	1
Learning Rate	0.4
Batch Size	50
Std of Weights	0.01
Epochs	300

Table 1: Optimized Parameters of Q1-part a

I calculated and plotted the mean squared and mean classification errors as follows:

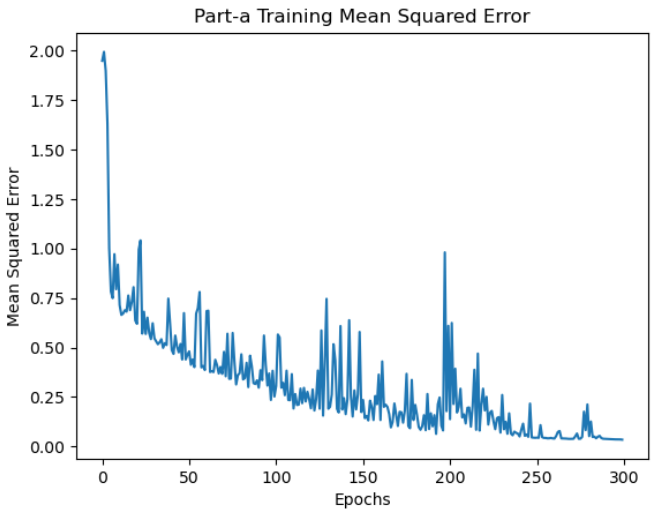

Figure 1: Training Mean Squared Error

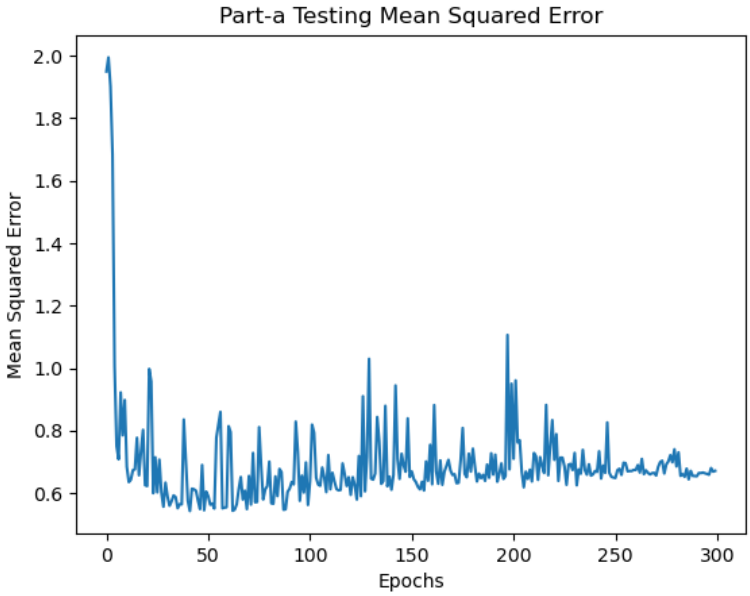

Figure 2: Testing Mean Squared Error

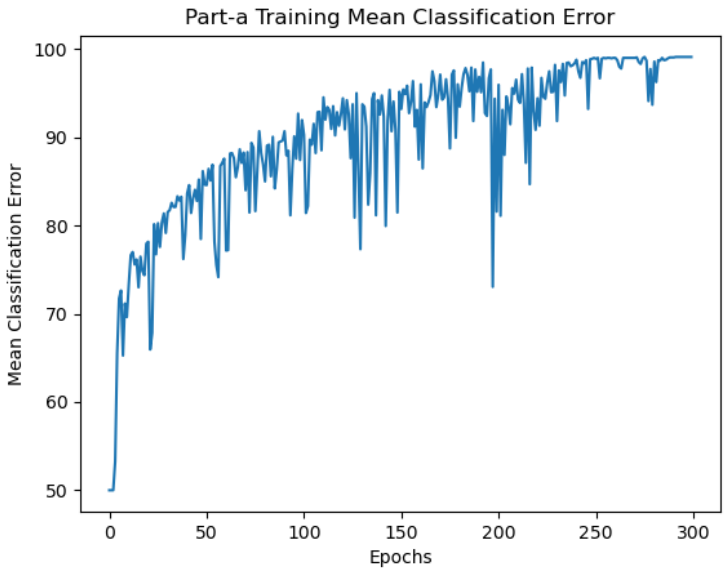

Figure 3: Training Mean Classification Error

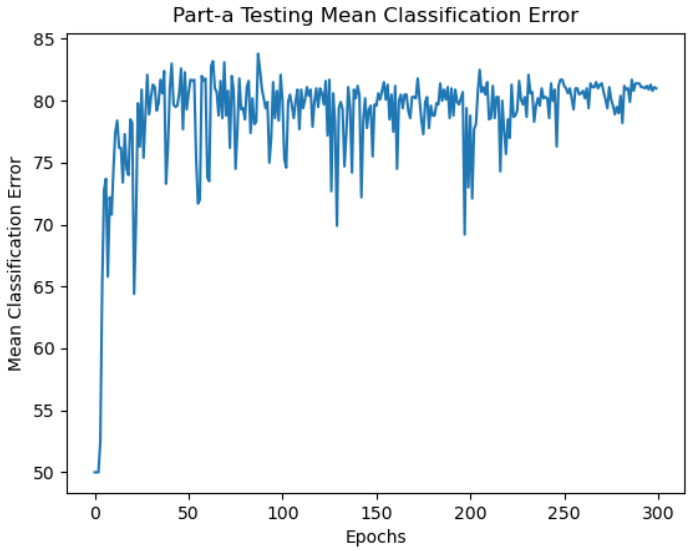

Figure 4: Testing Mean Classification Error

It can be seen that training MSE converges to zero as epoch increases, and testing MSE is converges to nearly between 0.6 and 0.7.

The training MCE converges to nearly 100% and testing MCE converges to 81%. This variance occur due to the variance in the data that the learning the data does not specifically mean perfect classification for the test data.

b) There is fast convergence in MSE plots due to Stochastic Gradient Descent. Figure 1 shows fast convergence that it converges around 200-250 epochs, which is a relative small epoch number. There occurs spikes and noise since the weights are characterized in batches and not individually. The training accuracy converges to nearly 100% and the testing accuracy converges around 81%. The training time also decreased in SGD because it does not propagate and update weights at each data point.

MSE may not be preferable for classification problems that there can be cases that classifications for the classes may be accurate but the predictions may be inaccurate and not converging to the ground truth. MSE may not show good succeeded results in these cases. Loss of information may occur after the classification of the predictions. MSE can be useful to measure the network in whether the algorithm is learning or not, but it is not an accurate error measurement for a classification problem.

c) In this part, the same classification problem is considered, but with a higher and lower number of hidden neurons. For the higher number of neurons, I chose the neuron number as 150, and for the lower I chose 5. Plots of MSE and MCE for training and test for the network with high, low and the optimal number of hidden neurons from part a is shown below:

Figure 5: Training Mean Squared Error for Optimal, High and Low Number of Hidden Neurons

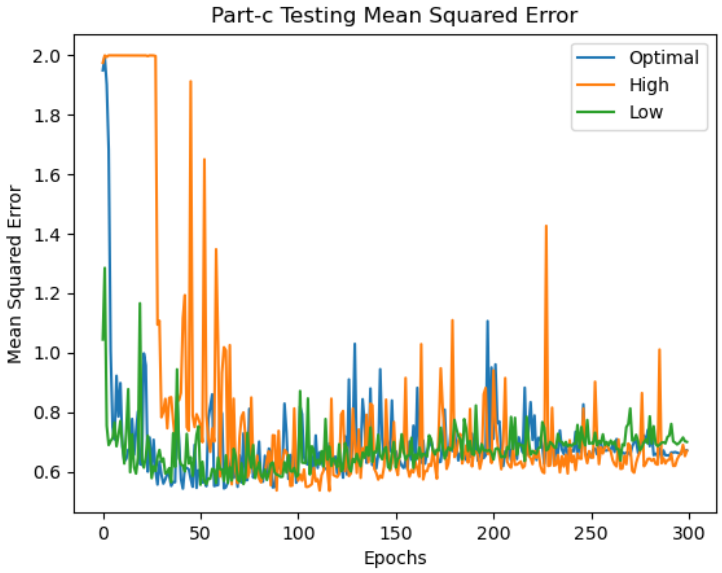

Figure 6: Testing Mean Squared Error for Optimal, High and Low Number of Hidden Neurons

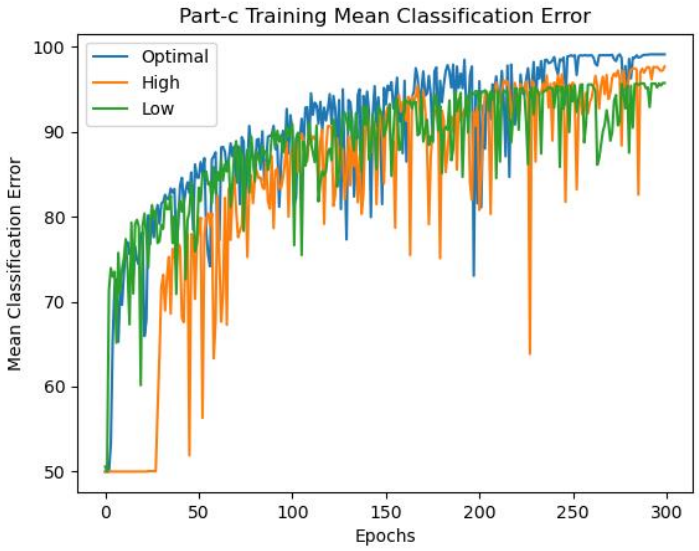

Figure 7: Training Mean Classification Error for Optimal, High and Low Number of Hidden Neurons

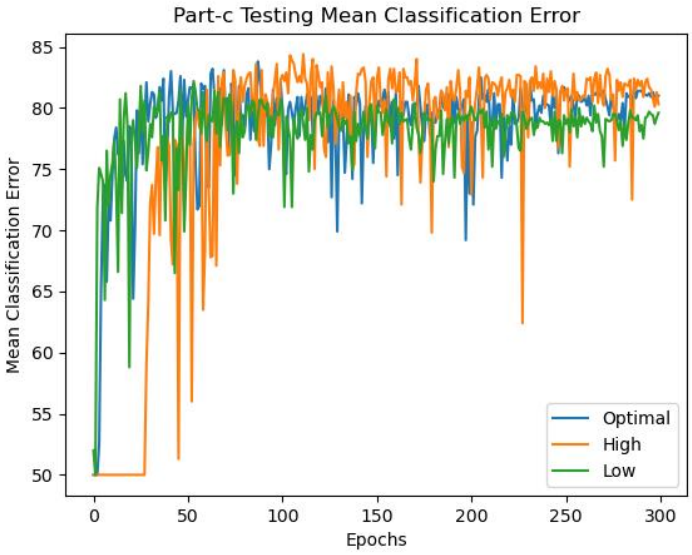

Figure 8: Testing Mean Classification Error for Optimal, High and Low Number of Hidden Neurons

It can be seen that the networks with higher and lower number of hidden units suffer more noise and their convergence is more later compared to the optimal case. Even their converged accuracies are less than the accuracy of optimal network. It can be caused by underfitting and overfitting cases for lower and higher number of hidden united networks respectively. If the hidden unit number is high, the oscillations increase due to overfitting of the past train. When the hidden unit number is low, the network will struggle classifying the train data and cannot predict the test data correctly. I did not want to choose the higher neuron number much higher because of the increased training time and processing power, therefore I selected it nearly 5 times of the optimal value. When we go right or left from the optimal point, the loss increases.

d) In this part, the network has two hidden layers instead of one. My optimized values for 2 hidden layer network is shown below:

Parameter	Value
Hidden Neuron 1st Layer	300
Hidden Neuron 2nd Layer	30
Output Neuron Number	1
Learning Rate	0.4
Batch Size	50
Std of Weights	0.01
Epochs	300

Table 2: Optimized Parameters of Q1-part d

I calculated and plotted the mean squared and mean classification errors as follows:

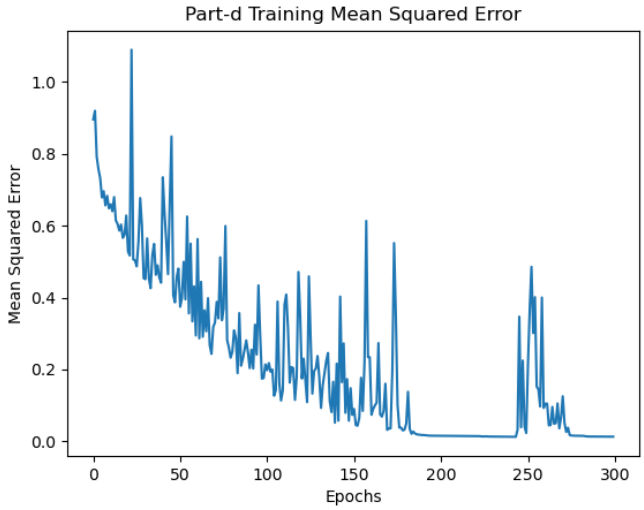

Figure 9: Training Mean Squared Error with Two Hidden Layers

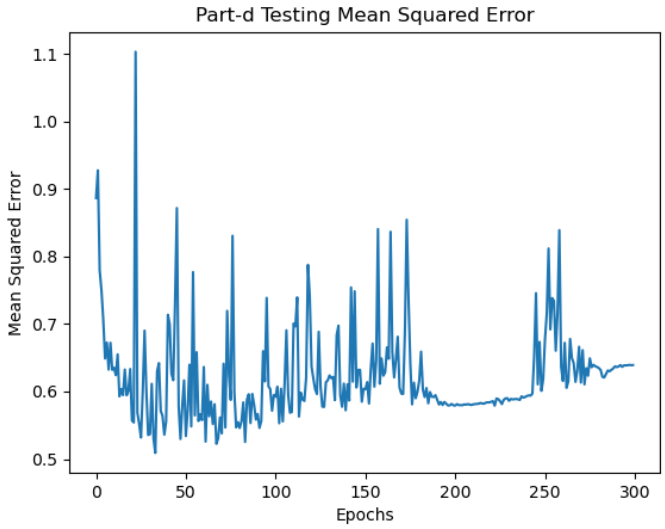

Figure 10: Testing Mean Squared Error with Two Hidden Layers

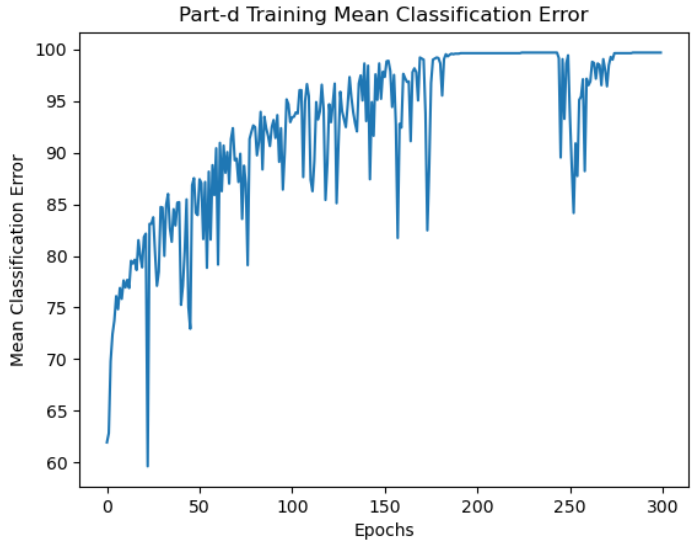

Figure 11: Training Mean Classification Error with Two Hidden Layers

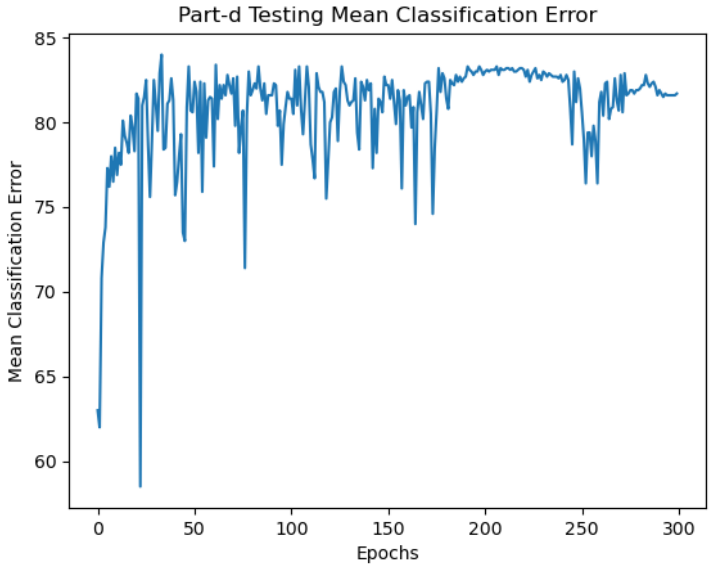

Figure 12: Testing Mean Classification Error with Two Hidden Layers

Comparing the results with part a, there is not a huge change between the learning curves among all metrics. All plots converges to the similar point with the ones in part a. The reason for this is the simplicity of the problem. Since the problem is not complex enough to require a network with 2 hidden layers, the training accuracy still converges nearly 100% but not reaches 100% exactly. Additionally for MSE, there are more spikes in two hidden layer network compared to the network in part a. For more complex problems, a network with

multiple hidden layers is required, but not in this case. Similar to the explanations in part c, if the number of layers increases, there may occur underfitting/overfitting.

e) Similar to the part d, in this part, still a network with 2 hidden layers is used, but additionally a momentum coefficient is used. The iterative momentum formula is shown below:

$$\triangle W(n) = -\eta \frac{\partial E}{\partial W} + \alpha \triangle W(n-1)$$

The parameters are same with the network in part d except the momentum coefficient, which I chose as $\alpha = 0.5$. The results are shown below:

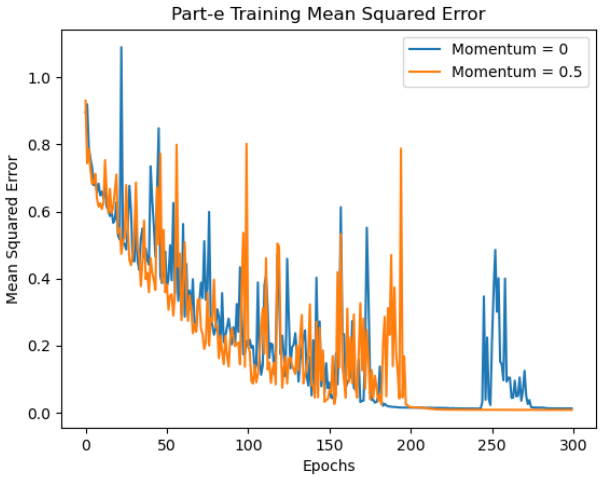

Figure 13: Training Mean Squared Error with Two Hidden Layers with and without Momentum

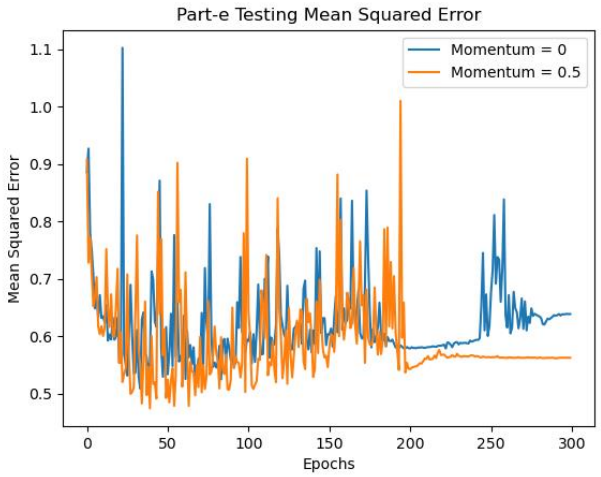

Figure 14: Testing Mean Squared Error with Two Hidden Layers with and without Momentum

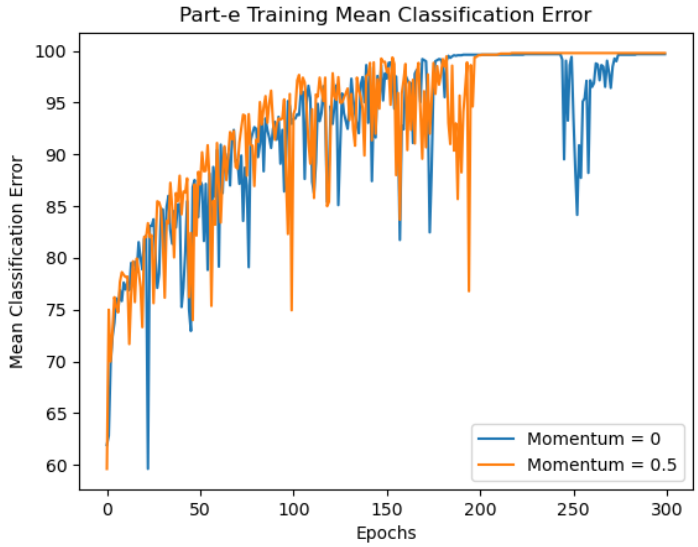

Figure 15: Training Mean Classification Error with Two Hidden Layers with and without Momentum

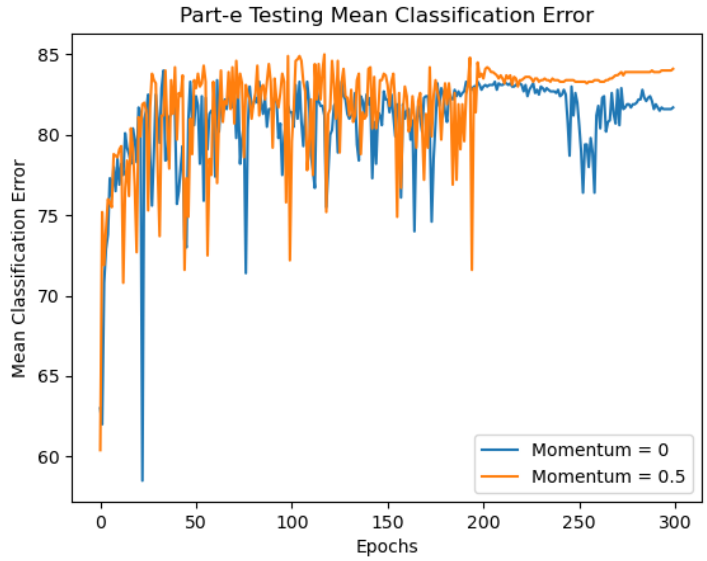

Figure 16: Testing Mean Classification Error with Two Hidden Layers with and without Momentum

In general, the momentum coefficient helps the network to find the global minimum, since the standard gradient descent algorithm may find the local minimum rather. It is prevented for the network to stuck in a local minimum. It helps in faster convergence.

For the results, it can be observed from the plots that with a momentum coefficient, network converges faster compared to the no momentum coefficient case. I chose the momentum coefficient as 0.5 to see the effect of it clearly. It provides a more stable

convergence. Additionally, the network with momentum has a higher classification accuracy, which is nearly 84%, and a lower mean square error for test data compared to the network with no momentum. However, since the task is simple enough as stated before, the effect of momentum in a network can be more clearly seen in a more complex problem. It is relatively less observable in this problem.

Question 2

In question 2, it is asked to implement a network that predicts the 4th Word from a given 3-word sequence. Sigmoid activation is used for the hidden layer and softmax function is used for the output. Their formulas are shown below:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{p} e^{z_j}}$$

where, p is the number of neurons in the layer.

Cross entropy error is used for loss function, whose formula is:

$$E_{cross\ entropy} = -\sum_{j=1}^{M} y_j \log\left(pred_j\right)$$

There are 250 words, so the p value is given as 250.

a) The desired parameters, but can be updated for better performance, are listed below:

- Mini batch size = 200
- $\eta = 0.15$
- $\alpha = 0.85$
- Max. Epoch size = 50
- Weight and bias std deviations = 0.01
- (D,P) = (32,256), (16,128), (8,64)

In the structure, a total 3 layers are used, 1 output and 2 hidden layers. I used a linear activation function (f(x) = x) for the embedding part and created another layer for it since it simply linearly projects the input to the output.

The methodology is inputting the samples and then concatenating them for the outputs for seperate results column-wise, then inputting it to the network.

The optimized parameters are shown at the table below:

Parameter	Value
Momentum Coefficient	0.85
Learning Rate	0.4
Batch Size	250
Std of Weights	0.1
Epochs	50

Table 3: Optimized Parameters of Q2

Results are shown below:

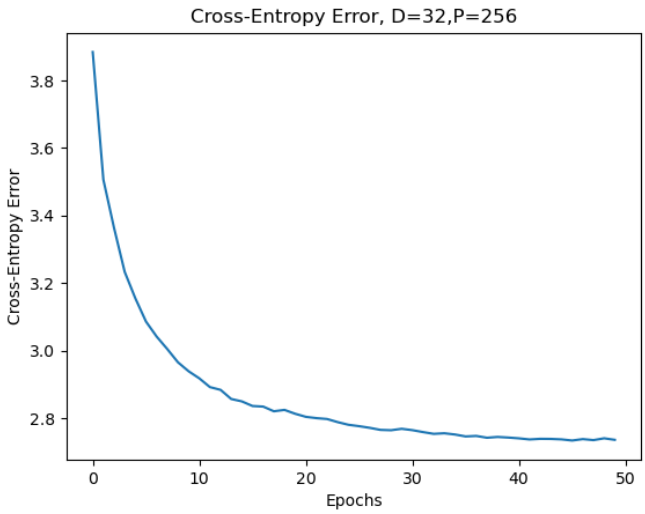

Figure 17: Cross-Entropy Error on the Validation Data when D = 32, P = 256

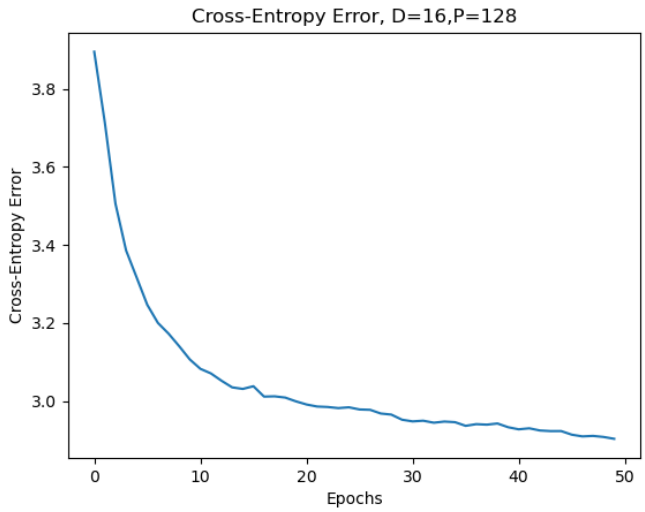

Figure 18: Cross-Entropy Error on the Validation Data when D = 16, P = 128

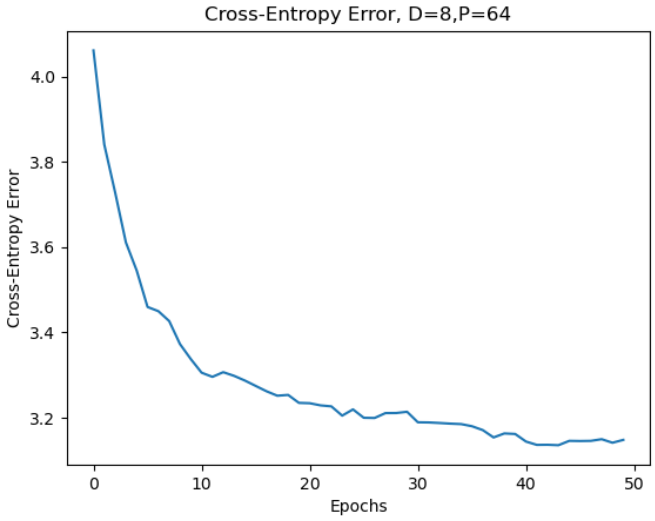

Figure 19: Cross-Entropy Error on the Validation Data when D = 8, P = 64

It can be observed that with more hidden units, the more the network performs better due to the complexity of the problem. The best performance is given at when D=32 and P=256, which has a cross-entropy error nearly 2.75. The errors and accuracies can be seen at the table below:

(D,P) Values	Cross-Entropy Loss	Accuracy
(8,64)	3.15	29.10%
(16,128)	2.90	32.54%
(32,256)	2.74	35.37%

Table 4: Cross-Entropy Losses and Accuracies for different (D,P) values

It is expected to have the better performance since due to the complexity of the problem and large size of the dataset, increasing the hidden unit size increases the performance.

b) In this part, it is asked to take 5 random test samples from the test data and forward it to the trained network to test the network. Afterwards, 10 predictions are executed to test whether they are accurate or not. The highest performed network, which has D = 32 and P = 256, is used for this purpose. The predictions can be seen below:

Phrases	Label	Predictions									
, he said	**.**	last	**.**	to	he	today	yesterday	:	,	?	for
put the other	**one**	in	i	way	,	two	people	day	**one**	team	.
you get on	**with**	it	.	your	him	you	**with**	this	?	the	and
nt want to	**see**	get	come	say	think	know	.	**see**	be	do	go
good in those	**days**	to	that	place	years	school	,	.	**days**	?	times

Table 5: Top 10 predictions of the 3 word phrases

For predictions, since the accuracy is around 30%, which is not a high accuracy, network slightly struggles while prediction. There are tons of possibilities for the 4th word of the sequence in terms of different areas and contexts. Since the network is trained with one possible outcome, when another possibility of outcome in a different context comes to the network, network fails to predict the correct label and predicts the word that it learned before. However, some predictions make sense and shows that even the accuracy is low, network still learns slightly. Additionally, some predicted words are actually occur as punctuations. Network could not differentiate the words and the punctuations. This is again because of the reason that network does not understand the context clearly and may predict the most repeated word in sequence, that may be the punctuations.

Although the predictions do not look consistent, in all 5 phrase tests, the network succeeded in finding the true label word in one of its 10 predictions. I bolded the true predictions among the 10 predicted values in the table 5. Therefore, as a result, network struggles to find the true labels exactly, but does learn and tries to make accurate predictions. This also may happen due to the input word amount, which is 3. It leaves less context for the network to learn the process.

Question 3

Inline questions for the notebooks are at the end of the report, inside the notebooks.

a) In this FullyConnectedNets Jupyter notebook, a fully connected network is implemented. It is executed by dividing the propagation steps. It leads to implementing the network on different applications. In the process, the forward propogation is the prediction step and outputs activation potentials are outputs. Gradients are found in backpropagation. These two steps are implemented for ReLU function. Afterwards, the sandwich layer is introduced. Then, the softmax and SVM loss functions are tested to confirm that the implementations are accurate and correct. A network with two layers is created by coding these various functions and tested by Solver class. Then, a 3 layer and a 5 layer networks are tested by overfitting 50 samples over 20 epochs. The update rule is implemented with Stochastic Gradient Descent, similar to our problems. Then, the update rules of RMSProp and Adam are introduced, which changes the learning rate for each parameter in the system. Finally, the network is trained on the CIFAR-10 dataset using the Solver class, which results in an accuracy of 51%.

b) In this Dropout notebook, it gives an introduction to dropout, which is a regularization technique used to avoid overfitting and individual learning among neurons. 1-p neurons become zero. The implementation of the dropout forward and backward modules is created

by creating a mask that creates a random array with same shape with input matrix. This matrix gives 1 if the matrix elements are less than p or 0 otherwise, then they are divided by p. In the training mode, 1-p terms are dropped out from the network, whereas in test mode all nodes are presented. Averages are also shown for different p values and how many terms are 0. Then, two fully connected networks with or without dropout are compared and accuracies are calculated and shown.

Appendix

Script That is Run in Command Prompt via "python ayberk_yarkin_yildiz_21803386.py x", For Both Question 1 and Question 2

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import h5py



def graph(x):
    plt.figure()
    plt.plot(x)
    plt.xlabel('Epochs')

def graphm(x,y,z):
    plt.figure()
    plt.plot(x)
    plt.plot(y)
    plt.plot(z)
    plt.legend(['Optimal', 'High', 'Low'])
    plt.xlabel('Epochs')

def graphd(x,y):
    plt.figure()
    plt.plot(x)
    plt.plot(y)
```

```
    plt.legend(['Momentum = 0','Momentum = 0.5'])
    plt.xlabel('Epochs')


def checkndim(x):
    if x.ndim == 1:
        x = x.reshape(x.shape[0],1)
    return x


def vec(x,y):
    nextt = np.zeros(y)
    nextt[x-1] = 1
    return nextt


def m1(x,y):
    nextt = np.zeros((x.shape[0],x.shape[1],y))
    for a in range(x.shape[0]):
        for b in range(x.shape[1]):
            nextt[a,b,:]=vec(x[a,b],y)
    return nextt


def graph2(x):
    plt.figure()
    plt.plot(x)
    plt.xlabel('Epochs')
    plt.ylabel('Cross-Entropy Error')


def m2(x,y):
    nextt = np.zeros((x.shape[0],y))
    for a in range(x.shape[0]):
```

```
        nextt[a,:]=vec(x[a],y)
    return nextt


def pb(x,y,z,dici,k):
    randoma = np.random.permutation(len(x))[0:5]
    tests = x[randoma]
    tests1 = m1(tests,dici)
    testsl = k[randoma]
    testsl = testsl.reshape(len(testsl),1)
    t10 = z.top10(tests1, 10)
    for i in range(5):
        print('[' + str(i+1) + ']' + str(y[tests[i,0]-1].decode('utf-8')) + '  ' + str(y[tests[i,1]-
1].decode('utf-8')) + '  ' + str(y[tests[i,2]-1].decode('utf-8')))
        print('True label= ' + str(y[testsl[i,0]-1].decode('utf-8')))
        stri = 'The Top-10 Candidates are: {'
        for j in range(10):
            stri += (str(y[t10[j,i]].decode('utf-8'))) + '  '
        print(stri+'}')



def q1():


    class Network:
        def __init__(self,dim,neuron,std,mean=0):
            self.dim = dim
            self.neuron = neuron
            self.bias = np.random.normal(mean,std,neuron).reshape(neuron,1)
            self.weight = np.random.normal(mean,std,dim*neuron).reshape(neuron,dim)
            self.param = np.concatenate((self.weight,self.bias), axis=1)
```

```
        self.chan = None
        self.err = None
        self.prevAct = None
        self.prevchan = 0

    def activation(self, x):
        if(x.ndim == 1):
            x = x.reshape(x.shape[0],1)
        self.prevAct = np.tanh(np.matmul(self.param, np.r_[x, [np.ones(x.shape[1])*-1]]))
        return self.prevAct

    def derAct(self,a):
        return 1-(a**2)

    def __repr__(self):
        return 'Input_Dim: '+str(self.dim)+', Neuron #: '+str(self.neuron)



class PropogateLayers:
    def __init__(self):
        self.lays = []

    def appendLayers(self,lay):
        self.lays.append(lay)
    def forward(self,imp):
        nextt = imp
        for layers in self.lays:
            nextt = layers.activation(nextt)
        return nextt
```

```
    def accur(self,x):
        print('Accuracy:', str(np.sum(x.predict(testimsf.T/255).T == testlbls)/len(testlbls)*100)
+ "%")


    def predict(self,imp):
        nextt = self.forward(imp)
        nextt[nextt>=0] = 1
        nextt[nextt<0] = -1
        return nextt


    def romando(self,imp,out):
        randindex = np.random.permutation(len(imp))
        imp,out = imp[randindex],out[randindex]
        return imp,out


    def BP(self,imp,out,lrate,batch,momentum=0):
        network_output = self.forward(imp)
        for k in reversed(range(len(self.lays))):
            layers = self.lays[k]
            if(layers == self.lays[-1]):
                layers.err = out - network_output
                layers.chan = (layers.derAct(layers.prevAct)) * layers.err


            else:
                nextlayers = self.lays[k+1]
                nextlayers.weight = nextlayers.param[:,0:nextlayers.weight.shape[1]]
                layers.err = np.matmul(nextlayers.weight.T, nextlayers.chan)
                layers.chan = (layers.derAct(layers.prevAct)) * layers.err
```

```
        for k in range(len(self.lays)):
            layers = self.lays[k]
            if (k==0):
                checkndim(imp)
                useimp = np.r_[imp,[np.ones(imp.shape[1])*-1]]
            else:
                useimp = np.r_[self.lays[k-1].prevAct,[np.ones(self.lays[k-1].prevAct.shape[1])*-
1]]

            if momentum == 0:
                layers.param = layers.param + (lrate*np.matmul(layers.chan, useimp.T))/batch
            else:
                ch = (lrate*np.matmul(layers.chan, useimp.T))
                layers.param = layers.param + momentum*layers.prevchan
                layers.prevchan = ch



    def mserr(self, imp, out):
        mse = np.mean((out.T - self.forward(imp.T))**2, axis=1)
        return mse

    def mcerr(self, imp, out):
        mce = np.sum(self.predict(imp.T) == out.T)/len(out)*100
        return mce

    def workout(self, imp, out, impT, outT, lrate, epoch, batch, momentum = 0):
        mse_err = []
        mce_err = []
        mset_err = []
```

```
        mcet_err = []
        for x in range(epoch):
            print('Epoch',x)
            imp,out = self.romando(imp, out)
            bat = int(np.floor(len(imp)/batch))

            for s in range(bat):
                self.BP(imp[batch*s:batch*(s+1)].T, out[batch*s:batch*(s+1)].T, lrate, batch)



            mse_err.append(self.mserr(imp, out))
            mce_err.append(self.mcerr(imp, out))
            mset_err.append(self.mserr(impT, outT))
            mcet_err.append(self.mcerr(impT, outT))

        return mse_err, mce_err, mset_err, mcet_err

    def __repr__(self):
        strr = ''
        for i, layers in enumerate(self.lays):
            strr += 'Layer ' + str(i) + ': ' + layers.__repr__() + '\n'
        return strr

filename = 'assign2_data1.h5'
f1 = h5py.File(filename,'r+')

testims = np.array(f1['testims'])
testlbls = np.array(f1['testlbls'])
trainims = np.array(f1['trainims'])
```

```
    trainlbls = np.array(f1['trainlbls'])


    testlbls = testlbls.reshape((len(testlbls),1))
    trainlbls = trainlbls.reshape((len(trainlbls),1))


    trainimsf = trainims.reshape(trainims.shape[0],(trainims.shape[1])**2)
    testimsf = testims.reshape(testims.shape[0], (testims.shape[1])**2)


    """ part a"""
    print('\nQuestion-1 Part-a')
    goo = PropogateLayers()
    goo.appendLayers(Network((trainims.shape[1])**2, 30, 0.01))
    goo.appendLayers(Network(30, 1, 0.01))


    print(goo)


    trainlbls[trainlbls==0] = -1
    testlbls = testlbls.astype(int)
    testlbls[testlbls == 0] = -1


    get_mse, get_mce, get_mset, get_mcet = goo.workout(trainimsf/255, trainlbls,
testimsf/255, testlbls, 0.4, 300, 50)


    goo.accur(goo)


    """ part c """


    print('\n\nQuestion-1 Part-c')
    print('High Hidden Neurons Training')
```

```
goohigh = PropogateLayers()
goohigh.appendLayers(Network((trainims.shape[1])**2, 150, 0.01))
goohigh.appendLayers(Network(150, 1, 0.01))
print(goohigh)

get_mseh, get_mceh, get_mseth, get_mceth = goohigh.workout(trainimsf/255, trainlbls, testimsf/255, testlbls, 0.4, 300, 50)

print('Low Hidden Neurons Training')
goolow = PropogateLayers()
goolow.appendLayers(Network((trainims.shape[1])**2, 5, 0.01))
goolow.appendLayers(Network(5, 1, 0.01))
print(goolow)

get_msel, get_mcel, get_msetl, get_mcetl = goolow.workout(trainimsf/255, trainlbls, testimsf/255, testlbls, 0.4, 300, 50)

""" part d """

print('\nQuestion-1 Part-d')
goohid = PropogateLayers()
goohid.appendLayers(Network((trainims.shape[1])**2, 300, 0.01))
goohid.appendLayers(Network(300, 30, 0.01))
goohid.appendLayers(Network(30, 1, 0.01))
print(goohid)

get_msehid, get_mcehid, get_msethid, get_mcethid = goohid.workout(trainimsf/255, trainlbls, testimsf/255, testlbls, 0.4, 300, 50)

goohid.accur(goohid)
```

```
    """ part e """

    print('\n\nQuestion-1 Part-e')
    goomom = PropogateLayers()
    goomom.appendLayers(Network((trainims.shape[1])**2, 300, 0.01))
    goomom.appendLayers(Network(300, 30, 0.01))
    goomom.appendLayers(Network(30, 1, 0.01))
    print(goomom)

    get_msemom, get_mcemom, get_msetmom, get_mcetmom =
goomom.workout(trainimsf/255, trainlbls, testimsf/255, testlbls, 0.4, 300, 50, 0.5)

    goomom.accur(goomom)

    """ plots """
    graph(get_mse)
    plt.title('Part-a Training Mean Squared Error')
    plt.ylabel('Mean Squared Error')
    plt.show()

    graph(get_mset)
    plt.title('Part-a Testing Mean Squared Error')
    plt.ylabel('Mean Squared Error')
    plt.show()

    graph(get_mce)
    plt.title('Part-a Training Mean Classification Error')
```

```
plt.ylabel('Mean Classification Error')
plt.show()

graph(get_mcet)
plt.title('Part-a Testing Mean Classification Error')
plt.ylabel('Mean Classification Error')
plt.show()

graphm(get_mse,get_mseh,get_msel)
plt.title('Part-c Training Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()

graphm(get_mset,get_mseth,get_msetl)
plt.title('Part-c Testing Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()

graphm(get_mce,get_mceh,get_mcel)
plt.title('Part-c Training Mean Classification Error')
plt.ylabel('Mean Classification Error')
plt.show()

graphm(get_mcet,get_mceth,get_mcetl)
plt.title('Part-c Testing Mean Classification Error')
plt.ylabel('Mean Classification Error')
plt.show()

graph(get_msehid)
```

```
plt.title('Part-d Training Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()

graph(get_msethid)
plt.title('Part-d Testing Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()

graph(get_mcehid)
plt.title('Part-d Training Mean Classification Error')
plt.ylabel('Mean Classification Error')
plt.show()

graph(get_mcethid)
plt.title('Part-d Testing Mean Classification Error')
plt.ylabel('Mean Classification Error')
plt.show()

graphd(get_msehid,get_msemom)
plt.title('Part-e Training Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()

graphd(get_msethid,get_msetmom)
plt.title('Part-e Testing Mean Squared Error')
plt.ylabel('Mean Squared Error')
plt.show()
```

```
    graphd(get_mcehid,get_mcemom)
    plt.title('Part-e Training Mean Classification Error')
    plt.ylabel('Mean Classification Error')
    plt.show()

    graphd(get_mcethid,get_mcetmom)
    plt.title('Part-e Testing Mean Classification Error')
    plt.ylabel('Mean Classification Error')
    plt.show()


def q2():


    class Network2:
        def __init__(self, dim, neuron, activ, std, mean=0):
            self.dim = dim
            self.neuron = neuron
            self.activ = activ
            if self.activ == 'sigmoid' or self.activ == 'softmax':
                self.bias = np.random.normal(mean,std,neuron).reshape(neuron,1)
                self.weight = np.random.normal(mean,std,dim*neuron).reshape(neuron,dim)
                self.param = np.concatenate((self.weight,self.bias), axis=1)
            elif self.activ == 'wordembed':
                self.d = neuron
                self.idim = dim
                self.weight = np.random.normal(mean,std,self.d*self.idim).reshape(self.d,self.idim)
            self.chan = None
            self.err = None
```

```
        self.prevAct = None
        self.prevchan = 0

    def activation(self,x):
        if (self.activ == 'sigmoid'):
            return np.exp(2*x)/(1+np.exp(2*x))
        elif (self.activ == 'softmax'):
            return np.exp(x-np.max(x))/np.sum(np.exp(x-np.max(x)),axis=0)
        elif (self.activ == 'wordembed'):
            return x

    def activations(self,x):
        if self.activ == 'sigmoid' or self.activ == 'softmax':
            if (x.ndim == 1):
                x=x.reshape(x.shape[0],1)
            self.prevAct = self.activation(np.matmul(self.param,np.r_[x, [np.ones(x.shape[1])*-
1]]))
        elif self.activ == 'wordembed':
            nextlayer = np.zeros((x.shape[0],x.shape[1], self.idim))
            for k in range(nextlayer.shape[0]):
                nextlayer[k,:,:] = self.activation(np.matmul(x[k,:,:], self.weight))
            nextlayer =
nextlayer.reshape((nextlayer.shape[0],nextlayer.shape[1]*nextlayer.shape[2]))
            self.prevAct = nextlayer.T
        return self.prevAct

    def derAct(self,a):
        if self.activ == 'sigmoid':
            return 2*(a*(1-a))
```

```
        elif self.activ == 'softmax':
            return a*(1-a)
        elif self.activ == 'wordembed':
            return np.ones(a.shape)

    def __repr__(self):
        return 'Input_Dim: '+str(self.dim)+', Neuron #: '+str(self.neuron)+ '\n Activation: '+
self.activ


class PropogateWords:
    def __init__(self):
        self.lays = []

    def appendLayers(self,lay):
        self.lays.append(lay)

    def forward(self,imp):
        nextt = imp
        for layers in self.lays:
            nextt = layers.activations(nextt)
        return nextt

    def predict(self,imp):
        nextt = self.forward(imp)
        if nextt.ndim == 1:
            return np.argmax(nextt)
        return np.argmax(nextt, axis=0)
```

```
    def romando(self,imp,out):
        randindex = np.random.permutation(len(imp))
        imp,out = imp[randindex],out[randindex]
        return imp,out

    def top10(self,imp,n):
        nextt = self.forward(imp)
        return np.argpartition(nextt, -n, axis=0)[-n:]

    def BP(self,imp,out,lrate,batch,momentum):
        network_output = self.forward(imp)
        for k in reversed(range(len(self.lays))):
            layers = self.lays[k]
            if layers == self.lays[-1]:
                layers.chan = out - network_output
            else:
                nextlayers = self.lays[k+1]
                nextlayers.weight = nextlayers.param[:,0:nextlayers.weight.shape[1]]
                layers.err = np.matmul(nextlayers.weight.T, nextlayers.chan)
                layers.chan = (layers.derAct(layers.prevAct)) * layers.err

        for k in range(len(self.lays)):
            layers = self.lays[k]
            if  k==0:
                checkndim(imp)
                useimp = imp
            else:
                useimp = np.r_[self.lays[k-1].prevAct, [np.ones(self.lays[k-1].prevAct.shape[1])*-
1]]
```

```
        if layers.activ == 'sigmoid' or layers.activ == 'softmax':
            ch = (lrate*np.matmul(layers.chan, useimp.T))/batch
            layers.param = layers.param + (ch + momentum*layers.prevchan)
        elif layers.activ == 'wordembed':
            chan3 = layers.chan.reshape((3,batch,layers.idim))
            useimp = np.transpose(useimp, (1,2,0))
            ch = np.zeros((useimp.shape[1], chan3.shape[2]))
            for a in range(chan3.shape[0]):
                ch = ch + lrate*np.matmul(useimp[a,:,:], chan3[a,:,:])
            ch = ch/batch
            layers.weight = layers.weight + (ch + momentum*layers.prevchan)
        layers.prevchan = ch

def workout(self,imp,out,impT,outT,lrate,epoch,batch,momentum):
    entlist = []
    for x in range(epoch):
        print('\nEpoch',x)

        imp,out = self.romando(imp, out)
        bat = int(np.floor(len(imp)/batch))

        for j in range(bat):
            batimp = m1(imp[batch*j:batch*(j+1)],d)
            batout = m2(out[batch*j:batch*(j+1)],d).T
            self.BP(batimp,batout,lrate,batch,momentum)

        ov = self.forward(impT)
        CEerr = -np.sum(np.log(ov)*outT.T)/ov.shape[1]
```

```
            print('Cross-Entropy Error: ',CEerr)
            entlist.append(CEerr)

            av = np.sum(self.predict(impT) == np.argmax(outT.T, axis=0))
            print('Correct: ',av)
            print('Accuracy: ', (av/ov.shape[1])*100,'%')

        return entlist

    def __repr__(self):
        strr = ''
        for i, layers in enumerate(self.lays):
            strr += 'Layer ' + str(i) + ': ' + layers.__repr__() + '\n'
        return strr



filename2 = 'assign2_data2.h5'
f2 = h5py.File(filename2,'r+')

words = np.array(f2['words'])
trainin = np.array(f2['trainx'])
trainout = np.array(f2['traind'])
valin = np.array(f2['valx'])
valout = np.array(f2['vald'])
testin = np.array(f2['testx'])
testout = np.array(f2['testd'])
```

Ayberk Yarkın Yıldız
21803386
EEE443 Project-2

```
""" part a"""
print('Question-2 Part-a')
p = 256
p2 = 128
p3 = 64
idim = 32
idim2 = 16
idim3 = 8
d = 250
lrate2 = 0.4
momentum2 = 0.85
batch2 = 250
epoch2 = 50

vin = m1(valin,d)
vout = m2(valout,d)



print('\nD = 32, P = 256')
goo2 = PropogateWords()
goo2.appendLayers(Network2(idim,d,'wordembed',0.1))
goo2.appendLayers(Network2(3*idim,p,'sigmoid',0.1))
goo2.appendLayers(Network2(p,d,'softmax',0.1))

err1 = goo2.workout(trainin,trainout,vin,vout,lrate2,epoch2,batch2,momentum2)

print('\nD = 16, P = 128')
goo22 = PropogateWords()
```

```
goo22.appendLayers(Network2(idim2,d,'wordembed',0.1))
goo22.appendLayers(Network2(3*idim2,p2,'sigmoid',0.1))
goo22.appendLayers(Network2(p2,d,'softmax',0.1))

err2 = goo22.workout(trainin,trainout,vin,vout,lrate2,epoch2,batch2,momentum2)

print('\nD = 8, P = 64')
goo23 = PropogateWords()
goo23.appendLayers(Network2(idim3,d,'wordembed',0.1))
goo23.appendLayers(Network2(3*idim3,p3,'sigmoid',0.1))
goo23.appendLayers(Network2(p3,d,'softmax',0.1))

err3 = goo23.workout(trainin,trainout,vin,vout,lrate2,epoch2,batch2,momentum2)

print('To execute the part b, after observation please close the plots')

graph2(err1)
plt.title('Cross-Entropy Error, D=32,P=256')
plt.show()

graph2(err2)
plt.title('Cross-Entropy Error, D=16,P=128')
plt.show()

graph2(err3)
plt.title('Cross-Entropy Error, D=8,P=64')
plt.show()
```

```
    """ part b"""

    pb(testin,words,goo2,d,testout)



question = sys.argv[1]

def ayberk_yarkin_yildiz_21803386_hw2(question):

    if question == '1' :
        q1()

    elif question == '2' :
        q2()

ayberk_yarkin_yildiz_21803386_hw2(question)
```

Question 3