Ayhan Okuyan
21601531

EEE443/543 Neural Networks - Assignment 2

Table of Contents

Ayhan Okuyan
21601531

Question 1

The question gives us a cost function given for a regression task which the engineer wants to run on a neural network, the cost function C_1 is given as follows.

$$C_1 = \frac{1}{2}\sum_n (y^n - \boldsymbol{w}^T x^n)^2$$

Here y^n corresponds to the true label of the n^{th} training sample and w corresponds to the weight matrix of the network layer. Thus, the term w^Tx is the prediction of the algorithm for the n^{th} training sample.

Part A

In this part, we are given a cost function separate from the given in the previous part and it asks for us to prove that minimizing this cost is equal to minimizing C_1. The cost function C_2 is given as follows.

$$C_2 = \frac{1}{2}\boldsymbol{w^T} A\boldsymbol{w} - b^T\boldsymbol{w}$$

We are asked to find the A and b that enables the same minimization with C_1. In order to accomplish this task, we first need to minimize C_1. This cost is a variant of the residual sum of squares cost the doesn't take the mean of the residual errors but instead dividing with two to neutralize the two factor that will come with taking the derivative of the cost function. In order to easily take the derivatives easily, we transform C_1 to a matrix notation as follows.

$$C_1 = \frac{1}{2}(\boldsymbol{Y} - \boldsymbol{XW})^T(\boldsymbol{Y} - \boldsymbol{XW})$$

Here, Y is the column vector with *(nx1)* dimensions and X is taken as the matrix with the transposes of each individual sample with p dimensions lined row-by-row with the pseudo -1 vector for the biases, hence its dimension become *(nx(p+1))*. Furthermore, W is the column vector with p dimensions that resembles weights with the dimensions with the added bias term that has the dimensionality *((p+1)x1)*. Then we expand this matrix product as follows.

$$C_1 = \frac{1}{2}\left(\boldsymbol{Y^T} - \boldsymbol{W^T X^T}\right)(\boldsymbol{Y} - \boldsymbol{XW})$$

$$C_1 = \frac{1}{2}(\boldsymbol{Y^T Y} - \boldsymbol{Y^T XW} - \boldsymbol{W^T X^T Y} + \boldsymbol{W^T X^T XW})$$

Then we can take the derivative with respect to W in order to find its extrema.

$$\frac{\partial C_1}{\partial \boldsymbol{w}} = \frac{1}{2}(-\boldsymbol{Y^T X} - \boldsymbol{Y^T X} + 2\boldsymbol{W^T X^T X})$$

Hence, we equate the derivative to zero and find w*.

$$\frac{1}{2}(-\boldsymbol{Y^T X} - \boldsymbol{Y^T X} + 2\boldsymbol{W^T X^T X}) = 0$$

$$-2\boldsymbol{Y^T X} + 2\boldsymbol{W^T X^T X} = 0$$

$$\boldsymbol{W^T X^T X} = \boldsymbol{Y^T X}$$

We take the transpose of both sides.

$$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{W} = \boldsymbol{X}^T\boldsymbol{Y}$$

$$\boldsymbol{w} = (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}^T\boldsymbol{Y} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}$$

In this derivation, we assume that the matrix X^TX is invertible. The ordinary least squares solution doesn't exist otherwise. After this is done, we apply the same logic to the second cost function, and we equate the found W values to each other in order to find A and b.

$$\frac{\partial C_2}{\partial \boldsymbol{w}} = \frac{A\boldsymbol{w} + A^T\boldsymbol{w}}{2} - b$$

Hence, we find the w vector as follows

$$\frac{A\boldsymbol{w} + A^T\boldsymbol{w}}{2} - b = 0$$

$$\frac{(A + A^T)\boldsymbol{w}}{2} - b = 0$$

$$\boldsymbol{w} = 2(A + A^T)^{-1}b$$

We should equate the two w vectors to each other and find the relationship between the two since they must be equal, however in order to do that, we should assume that A is symmetric. We will check the validity of this statement after we have found A.

$$\boldsymbol{w} = (A)^{-1}b = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}$$

Here, we can see the relation between the terms and can write A and b as follows.

$$A = \boldsymbol{X}^T\boldsymbol{X}$$

$$b = \boldsymbol{X}^TY$$

We can clearly see that the presumptions that we have made about A is true since X^TX always yields to a symmetric matrix. Then, we are asked to find an update rule for the gradient descent algorithm and for that we ill use the derivative that we have found in this part and can write the gradient descent update as follows.

$$\boldsymbol{w}_{t+1} \coloneqq \boldsymbol{w}_t - \eta\Delta\boldsymbol{w}_t$$

Here, η is the learning rate hyperparameter for the designed algorithm. In this case, the update rule evolves into the following.

$$\boldsymbol{w}_{t+1} \coloneqq \boldsymbol{w}_t - \eta(A\boldsymbol{w}_t - b) = \boldsymbol{w}_t - \eta((\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w}_t - \boldsymbol{X}^TY)$$

Part B

The second part of the question we are asked to find an appropriate change of variables that makes the update rule from the previous part equal to the update of the weights that come from the cost function C_3 given below.

$$C_3 = \frac{1}{2}\tilde{\boldsymbol{w}}^{\boldsymbol{T}}A\tilde{\boldsymbol{w}}$$

In this part, we also follow the same procedure and take the derivative of the cost function which outputs the following.

$$\frac{\partial C_3}{\partial \widetilde{\boldsymbol{w}}} = \frac{1}{2}(A^T\widetilde{\boldsymbol{w}} + A\widetilde{\boldsymbol{w}})$$

From the previous part, we already know that A matrix is symmetric and equal to X^TX, hence we can rewrite the expression as follows.

$$\frac{\partial C_3}{\partial \widetilde{\boldsymbol{w}}} = \frac{1}{2}2(A\widetilde{\boldsymbol{w}}) = A\widetilde{\boldsymbol{w}}$$

Then, we can solve this problem by rearranging the gradient of the second cost function in the following form running the same invertible assumption that we have used on A.

$$\frac{A\boldsymbol{w} + A^T\boldsymbol{w}}{2} - b = A\boldsymbol{w} - b = A\boldsymbol{w} - AA^{-1}b$$

Hence, we can say that the gradient is as given below.

$$\frac{\partial C_2}{\partial \boldsymbol{w}} = A(\boldsymbol{w} - A^{-1}b)$$

Then we can equate the two gradients as given

$$A(\boldsymbol{w} - A^{-1}b) = A\widetilde{\boldsymbol{w}}$$

Here we can clearly see the change of parameters requested from us as

$$\widetilde{\boldsymbol{w}} = \boldsymbol{w} - A^{-1}b$$

Part C

This part asks us to determine the maximum learning rate η for the weight updates for a stable solution assuming that A is symmetric and positive-definite. Since A is symmetric and positive-definite, we can say that the solution space of A is convex. This result says that there is a global minimum that minimizes the solution.

Since A is positive-definite and symmetric, its singular values coincide with its eigenvalues. This further implies that the Hessian matrix of A is 2A. The further analysis will be using the eigenvalues of the Hessian(A), which corresponds to the eigenvalues of 2A, hence two times the eigenvalues of A.

Assuming we have linear convergence, there exist some intervals where the gradient descent algorithm converges to the global minimum. Furthermore, we will be defining these intervals' convergence upper bounds instead of using the convergence rates that are defined in terms of the condition number (condition number is given as the division of the maximum eigenvalue with the minimum eigenvalue).

Convergence to the global minimum is observed only on the following learning rates defined in terms of the maximum and minimum eigenvalues.

Ayhan Okuyan
21601531

Learning Rate	Upper-Bound on Convergence Rate	Worst Case Convergence Rate
$\eta < \frac{2}{\lambda_{max}}$	$\lvert\eta * \lambda_{max} - 1\rvert^2$	$\lvert\eta * \lambda_{max} - 1\rvert$
$\eta = \frac{2}{\lambda_{max} + \lambda_{min}}$	$\left(\frac{\epsilon - 1}{\epsilon + 1}\right)^2$	$\frac{\epsilon - 1}{\epsilon + 1}$
$\eta = \frac{1}{\lambda_{max}}$	$\left(\frac{\epsilon - 1}{\epsilon}\right)^2$	$\frac{\epsilon - 1}{\epsilon}$

Table 1 – Convergence Rates for Learning Rate Intervals [1]

Here we observe that

$$\frac{2}{\lambda_{max}} > \frac{1}{\lambda_{max}} and \ \frac{2}{\lambda_{max}} > \frac{2}{\lambda_{max} + \lambda_{min}}$$

The question asks us to find the greatest learning rate for stable convergence, hence we can define the learning rate as follows.

$$\eta_{max} = \lim_{\propto \to 1^-} \propto \frac{2}{\lambda_{max}}$$

Ayhan Okuyan
21601531

Question 2

In this question, we are asked to construct a feed-forward neural network structure for a binary classification task. The task is to classify the given 32x32 images to two categories, either “cat” or “car” and we will be using a stochastic mini-batch gradient descent optimization and will be using two separate error metrics given as mean squared error and mean classification error. Mean classification error is described as the “percentage of correctly classified images” and this implementation is done according to this definition of the term and mean squared error is as follows.

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

Here, N is the number of total samples in either the training or test data since we are requested to keep the statistics for both and i represents the i^{th} sample of the corresponding data.

Part A

This part requires us to implement a neural network with one hidden layer and one output layer with the use of the backpropagation algorithm until convergence. The question implicitly has two phases that should be considered separately one being the framework construction of the network and the other as parameter optimization.

Framework Construction

The first part is to create a framework for the neural network. On my own volition, I have decided to implement a semi-generic class-based implementation that uses Python’s object-oriented programming side. The first part that I have implemented is the Layer class which is given and discussed below.

Layer Class

```
class Layer:
    def __init__(self, inputDim, numNeurons, std, beta=1, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons

        self.beta = beta

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numN
eurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)

        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

    def activtanh(self, x):
        if(x.ndim == 1):
            x = x.reshape(x.shape[0],1)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def tanhDer(self, x):
        return self.beta*(1-(x**2))
```

This structure provides a shell that keeps in memory the weights and biases of the layer together with its last activation, delta, error. The weights and biases are initiated with Gaussian random

variables with zero mean. The standard deviation of the distribution is asked from the user as a hyperparameter to be optimized. The activation function that we were required to implement in this section was the tanh activation, and I have only implemented that function and its derivative and that is the reason why I have addressed the framework as semi-generic. The tanh activation is given below, however for the implementation, I have used the built in numpy.tanh function.

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

For the activation function, I have considered the derivation below. We have concatenated the weight and bias vectors in the columns to obtain the matrix referred to as W_E (referred as self.weightsE in the code) and at the same time we have added a -1 row to the input matrix whose rows are the features and the columns that corresponds to the samples. After taking the matrix multiplication of those, I have passed it though tanh activation and returned the result.

$$y_{layer} = \tanh(v_{layer})$$

$$v_{layer} = \boldsymbol{W_E} * \tilde{X}$$

$$where\ \boldsymbol{W_E} = [\boldsymbol{W} \mid \theta]\ , \tilde{X} = \begin{bmatrix} X \\ -1 \end{bmatrix}$$

After implementing the activation, I have implemented the derivative of the activation since we need to calculate the derivative of the v matrices of the layer for the backpropagation algorithm to work properly. In order to do that I have derived the derivative of the tanh function which turns out to be as follows.

$$\frac{\partial \tanh\,(x)}{\partial x} = \operatorname{sech}^2(x) = 1 - \tanh^2(x)$$

This result turns out to be highly convenient since the derivative involves the output of the layer since we can describe it as,

$$\frac{\partial \tanh\,(V_{layer}(X))}{\partial X} = 1 - \tanh^2(V_{layer}) = 1 - Y_{layer}^2$$

In the code segment, I have also used a beta parameter to scale the inputs in the case of an overflow if needed, however, after the optimization I have recognized that there is no need for such scaling hence beta parameters through this question has been explicitly placed as 1.

This is the reasoning behind saving the last activations of the neurons and not the v matrices and it is computationally efficient since we are already calculating the activated versions in the forward propagation. After the layer is constructed, I have moved to the MLP class which uses the combination of the Layer instances to operate the forward and backpropagation algorithms.

MLP Class

The constructor is instantiated only with an empty list called the layers and adds to that layer with the help of the addLayer function which are given below.

```
class MLP:
    def __init__(self):
        self.layers = []

    def addLayer(self, layer):
        self.layers.append(layer)
```

Since we are using a list structure, we will be using an iteration loop to move through one layer to another. After that I have implemented the forward propagation that uses a for loop to iteratively calculate the activations since the added layer structures create a nested calculation pattern.

$$y_o = \tanh\left(\boldsymbol{W_{Eo}} * \ldots \tanh\left(\boldsymbol{W_{E2}} * \tanh\left(\boldsymbol{W_{E1}} * \tilde{X}\right)\right)\right)$$

Hence the code becomes,

```
    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activtanh(out)
        return out
```

After that, I have implemented the backpropagation algorithm for the network using the delta rule since individually computing every derivative separately for each layer and each weight becomes computationally intractable as the number of weights increase.

When we look at the output layer, since we are using the mean squared error loss, we can write the layer error and its derivative as follows.

$$Loss_{MSE} = \boldsymbol{E} = \frac{1}{2}\left(Y - \hat{Y}\right)^2$$

$$e_{output} = \frac{\partial \boldsymbol{E}}{\partial \hat{Y}} = \hat{Y} - Y$$

Here, Y represents that labels of the training samples and Y_hat represents the result of the network, meaning the activation of the output layer. Then we calculate the delta of the output layer which can be expressed as follows.

$$\delta_{output} = \frac{\partial \tanh\left(V_{output}(X)\right)}{\partial X} \odot \frac{\partial \boldsymbol{E}}{\partial Y} = \frac{\partial \tanh\left(V_{output}(X)\right)}{\partial X} \odot e_{output}$$

$$\delta_{output} = (1 - Y_{layer}^2) \odot (\hat{Y} - Y)$$

This concludes the calculations of the output layer, then we move to finding the error and delta for the hidden layers. The calculations are as follows where the components with the prime sign represent the delta and weights of the next layer from input to output.

$$e_{hidden} = \boldsymbol{W'^T} * \delta'$$

$$\delta_{hidden} = \frac{\partial \tanh(V_{hidden}(X))}{\partial X} \odot e_{hidden} = (1 - Y_{hidden}^2) \odot (\boldsymbol{W'^T} * \delta')$$

Take into consideration that this analysis is done according to the fact that network receiving a batch (mini batch) of inputs in parallel. Hence the derivations that has gone through lecture is modified with the dot products for the deltas. Hence, we now need to consider the update rules for the output and hidden layer structures. This rule is generalized for all the layers with specifying the inputs that will be used for each algorithm. Here, the prime sign is used for the updated weights.

$$\boldsymbol{W'} \coloneqq \boldsymbol{W} + \eta * \delta * \ddot{X}^T$$

$$where, \qquad \ddot{X} = \begin{cases} \tilde{X}\ if\ output \\ \dot{Y}'\ if hidden \end{cases}$$

$$\dot{Y}' = \begin{bmatrix} \dot{Y} \\ -1 \end{bmatrix}, \qquad where\ \dot{Y}\ represents\ the\ output\ of\ the\ previous\ layer's\ activation$$

Then, we have implemented the code segment of the backpropagation algorithm as follows. Furthermore, we divide the update term to the batch size so that the neurons do not saturate.

```
    def backProp(self, inp, out, lrnRate, batchSize):

        net_out = self.forward(inp)
        #print('network out: ', net_out.shape)

        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]

            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.error = out - net_out
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            #print(inp)
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = np.r [inp, [np.ones(numSamples)*-1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*-1]]
            lyr.weightsE += (lrnRate* np.matmul(lyr.delta, inputToUse.T))/batchSize
```

Then we have written the simple function to receive the predictions of the network for error calculation in the training phase. This part is also implementation specific since we have used the tanh activation whose range is bounded between -1 and 1, I have used a risk avert zero thresholding to make the outputs that are negative to be equal to -1 and the positives to be 1, since those are the values that the algorithm should be trying to reach with the mean squared error loss. For the implementation, I have chosen to use one output neurons instead of two for the following two reasons:

- The activation function is tanh
- The task is binary classification

As a notice for further implementation, I have also manipulated the data labels that were given as zeros and ones to ones and minus ones. The code is given below.

```
    #only for binary classificaiton with 1 output neurons
    def prediction(self, inp):
        out = self.forward(inp)
        out[out>=0] = 1
        out[out<0] = -1
        return out
```

Since we have all the components needed for the construction of the neural network, we could now explicitly create a function for the training with mini batches this part uses the backProp function in order to iterate the network over mini-batches that are taken from the shuffled training data,

calculates the errors, adds them to their respective lists, and returns the errors after the training is done.

```
    def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize):
        mseList = []
        mceList = []
        mseTList = []
        mceTList = []

        for ep in range(epochNum):
            print('Epoch', ep)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]

            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                self.backProp(inp[batchSize*j:batchSize*j+batchSize].T,\
                              out[batchSize*j:batchSize*j+batchSize].T,\
                              lrnRate, batchSize)

            mse = np.mean((out.T - self.forward(inp.T))**2, axis=1)
            mseList.append(mse)
            mce = np.sum(self.prediction(inp.T) == out.T)/len(out)*100
            mceList.append(mce)
            mseT = np.mean((outTest.T - self.forward(inpTest.T))**2, axis=1)
            mseTList.append(mseT)
            mceT = np.sum(self.prediction(inpTest.T) == outTest.T)/len(outTest)*100
            mceTList.append(mceT)
        return mseList, mceList, mseTList, mceTList
```

This implementation chooses to run according to the number of batches and hence, in the parameter optimization phase, I have chosen batch numbers that the number of training samples are divisible to. This concludes the creation of the framework for the question. The next part is to run the algorithm to train, optimize and see the results.

Optimization and Review of the Results

Here, we simply construct the net and use the train function and plot the errors. The code is as follows.

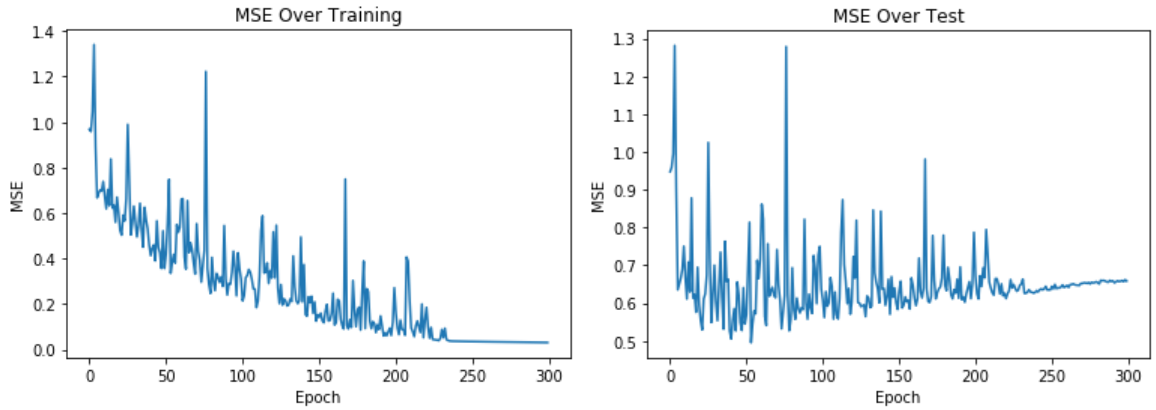
```
neuralNet = MLP()
neuralNet.addLayer(Layer(inputWH**2, 18, 0.02, 1))
neuralNet.addLayer(Layer(18, 1, 0.02, 1))

print(neuralNet)

train_labels[train_labels == 0] = -1

test_labels = test_labels.astype(int)
test_labels[test_labels == 0] = -1

#0.3 lr
#300 epoch
mses, mces, mseTs, mceTs = neuralNet.train(train_img_flat/255, train_labels, test_i
mg_flat/255, test_labels, \
                                           0.35, 300, 38)
```

Here, we observe that the images are flattened and normalized since this is an MLP algorithm and after length optimization, the best results are achieved with the following parameters.

Hyper-Parameter	Selected Value
Hidden Neurons	18
Output Neurons	1
Learning Rate	0.35
Batch Size	38
Weights Std	0.02
Epoch Number	300

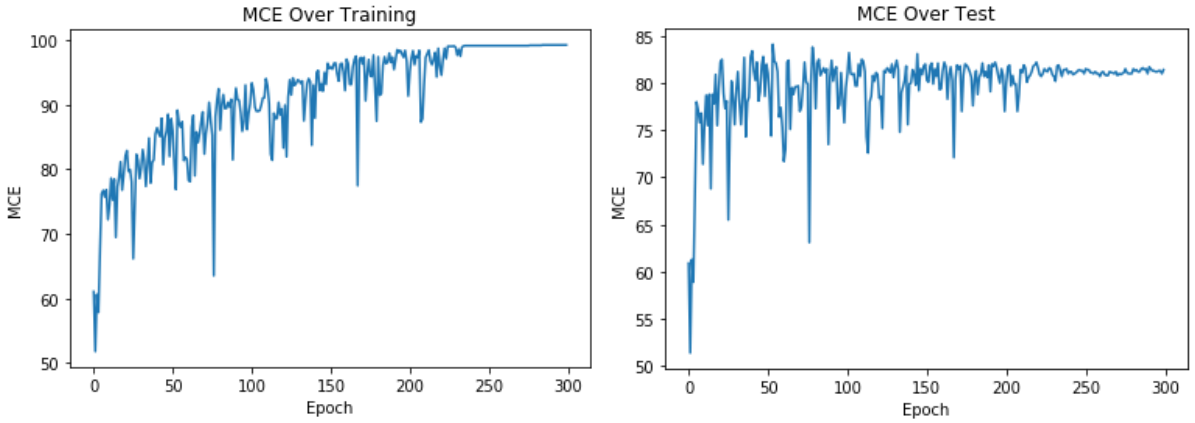
Table 2 – The Optimized Parameters for Question 2-A

The error graphs for the test and training MSEs are given below.



Figures 1-2 – Training and Test Mean Squared Errors over Training Epochs

Here, we observe that the MSE over training converges to zero, meaning the algorithm memorizes the training data and here we can see the MSE stabilizing over the 0.6 range. The MCE errors (accuracies) are also given below.



Figures 3-4 – Training and Test Accuracies over Training Epochs

Similar with Figure 1, the algorithms, classification accuracy over the training converges to 100% while the test accuracy converges to 81.39%. This is due to the difference of variance in the data that learning training data perfectly doesn't mean perfect classification for the test data.

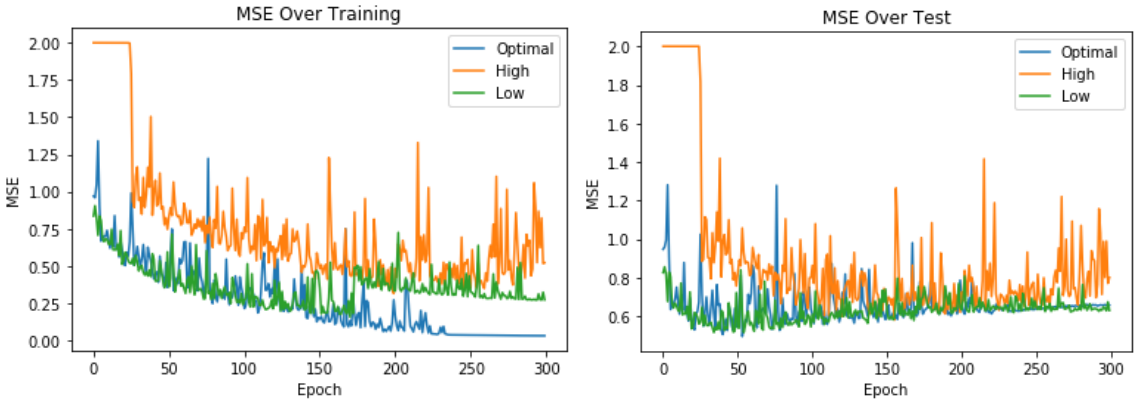
Part B

This part asks us to resolve if the mean squared error or the mean classification is an adequate error for a classification task. Here, we see that the MSE is inversely proportional to MCE, however, since

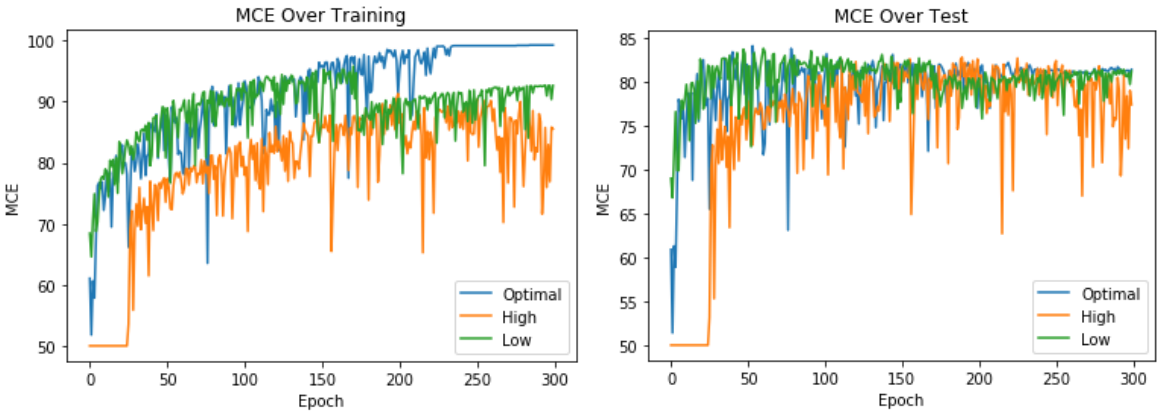
MSE considers the outputs of the network without the thresholding, there can be cases where nearly all of the classifications for all of the classes are accurate but one whose predicted value is far from the ground truth. In those types of cases, the classification can be considered, however the MSE would not enable us to see this success. By looking at the graphs 1-4, we can observe that there is strong correlation between the MCE and MSE, since MSE dropping would eventually lead to an increase in the correctly classified classes. However, we can say that while MSEs can be useful in terms of understanding if the algorithm is learning or not, it is not an adequate error metric for a classification task.

Part C

This part asks us to run the same classification task, this time with two other number of hidden neurons and one substantially high and one substantially low with respect to the number of hidden neurons we have optimized in Part A, which is 18. For the higher learning rate, I have chosen 800 and for the low I have chosen 5. The plots of the MSEs and MCEs for training and test are given below.



Figures 5-6 - Training and Test Mean Squared Errors over Training Epochs for the Networks with Optimal, High and Low Number of Hidden Units in the Hidden Layer



Figures 7-8 - Training and Test Classification Accuracies over Training Epochs for the Networks with Optimal, High and Low Number of Hidden Units in the Hidden Layer

Here, we can observe that the MSE on both the training and test data, the high and low algorithms are substantially higher than the errors of the optimal algorithm. Since the optimal result I have achieved was 18, the low number of 5 is closer to the optimal algorithm. When looked at the classification accuracies, the high and low algorithms learn less than the optimal algorithm, however,

when we observe the test data, the differences are non-substantial, meaning that the memorization of the training data doesn't have a substantial effect on the test classification. On the contrary, the optimal rates are a tad bit higher than the other two algorithms. This is due to the topology of the data which we understand is easily learnable. The other factor is that the learning rate that I have chosen is higher than the normal learning rates allowing the non-optimal algorithms to pass over some local minima and be on par with the optimal algorithm.

Part D

This part of the question requires us to run the same classification task using two hidden layers instead of the one that we have done on Part A. As explained, I have written a semi-generic framework for any number of hidden layers. Hence, I have used the same class structure, and trained the network as given below.

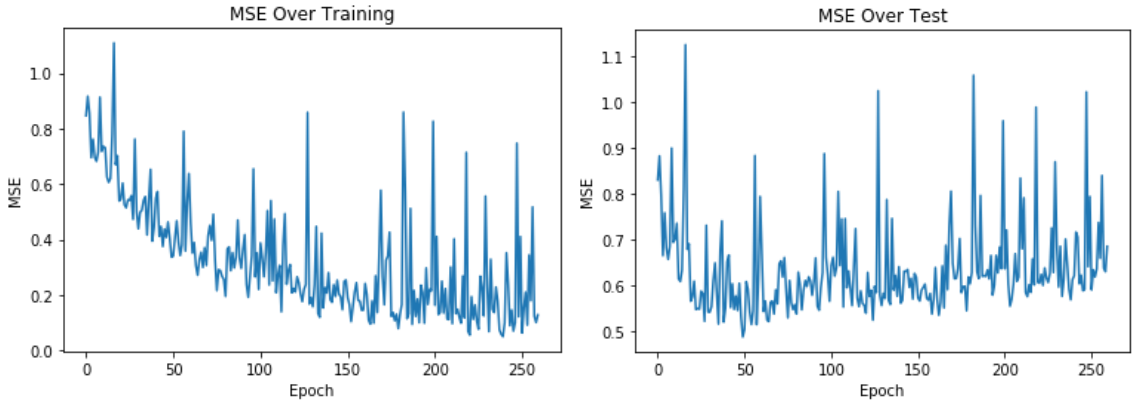
```
neuralNetTwoHidden = MLP()
neuralNetTwoHidden.addLayer(Layer(inputWH**2, 450, 0.02, 1))
neuralNetTwoHidden.addLayer(Layer(450, 45, 0.02, 1))
neuralNetTwoHidden.addLayer(Layer(45, 1, 0.02, 1))

print(neuralNetTwoHidden)

mses2, mces2, mseTs2, mceTs2 = neuralNetTwoHidden.train(train_img_flat/255,\
                               train_labels, test_img_flat/255, test_labels,\
                                                    0.4, 260, 38)
```

Here, after the optimization, we have achieved the optimal case on the parameters given below.

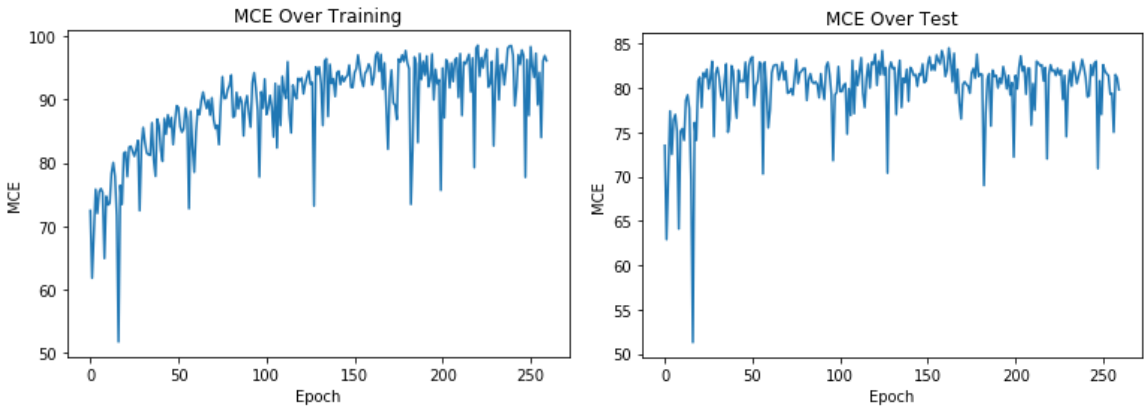
Hyper-Parameter	Selected Value
Hidden Neurons 1st Layer	450
Hidden Neurons 2nd Layer	45
Output Neurons	1
Learning Rate	0.4
Batch Size	38
Weights Std	0.02
Epoch Number	260

Table 3 - The Optimized Parameters for Question 2-D

The accuracy of the algorithm has become around 80%. The error plots are given below.



Figures 9-10 - Training and Test Mean Squared Errors over Training Epochs with Two Hidden Layers



Figures 11-12 - Training and Test Accuracies over Training Epochs with Two Hidden Layers

Here, I have observed by looking at the difference between the network in Part A and the network in Part D, there is an important difference. In the two-layer network, the network has become too complex for the algorithm to learn the pattern in the training as good as the other algorithm. Here, we see that the complexity of the model has surpassed the complexity of the data, since the training accuracy couldn't reach 100% as the algorithm in Part A did. In my implementation, the one-layer network has been better in term of both the MSE and MCE. Here, we can observe that the test approaches convergence faster, and this was expected since deeper the network, we usually expect faster learning.

Part E

This part is nearly the same with Part D however, this time we are asked to update the weights using a momentum coefficient. In a gradient descent algorithm, learning rate is generally used very little since as the learning rate increases, the updates will oscillate when approaching a minimum. However, this will result with a very slow learning process. The motivation behind the momentum coefficient is to change the direction of the update along the previous updates to reduce the oscillations. Momentum update for a weight can be generally written as follows for a gradient descent update on a generic update time t where n is the most recent update that will be done.

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

It would be computationally costly to save all the previous gradients in memory; hence, we instead apply this iterative formula. Here, we induce the term last update of the layer as a separate term and iteratively update that term.

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} + \alpha \Delta w_{ji}(n-1)$$

Hence, we have changed the following parts of the code that we have written in the previous section to meet the criteria.

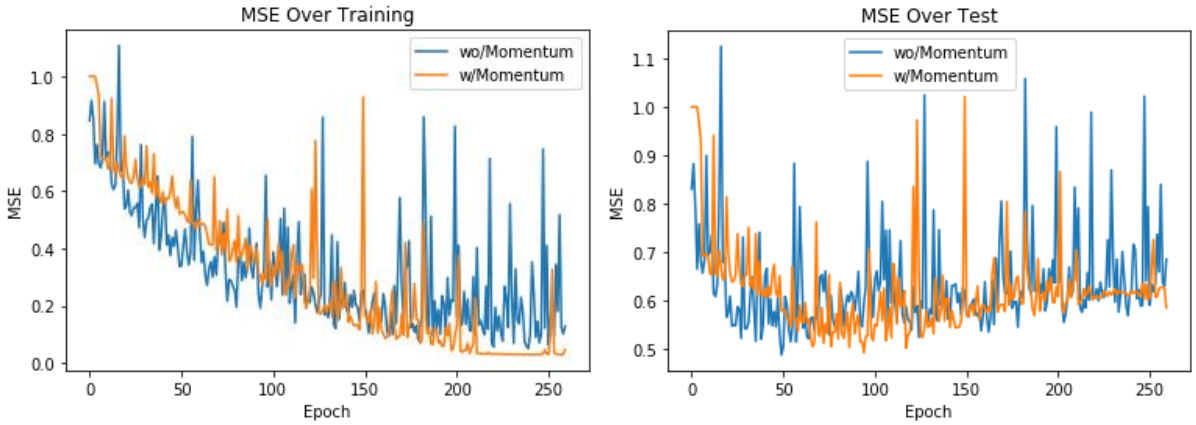
```
    def backProp(self, inp, out, lrnRate, momCoeff, batchSize):

        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]

            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.error = out - net_out
```

```
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = np.r_[inp, [np.ones(numSamples)*-1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*-1]]
            update =  lrnRate * np.matmul(lyr.delta, inputToUse.T)
            lyr.weightsE += update + momCoeff * lyr.prevUpdate
            lyr.prevUpdate = update
```

The altered section is highlighted with red. Furthermore, we construct and train the network and run it as follows with a momentum coefficient of 0.1, which is admissible since it is required to be between 0.1 and 0.5.

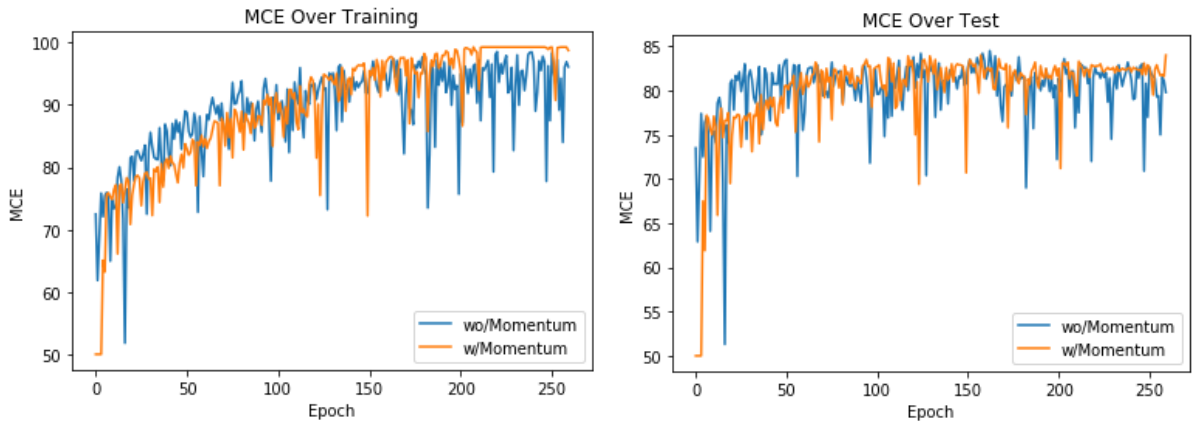
```
neuralNetTwoHiddenM = MLPwMom()
neuralNetTwoHiddenM.addLayer(LayerwMom(inputWH**2, 450, 0.02))
neuralNetTwoHiddenM.addLayer(LayerwMom(450, 45, 0.02))
neuralNetTwoHiddenM.addLayer(LayerwMom(45, 1, 0.02))

momentum_coefficient = 0.1

mses2M, mces2M, mseTs2M, mceTs2M = neuralNetTwoHiddenM.train(train_img_flat/255, \
                                   train_labels, test_img_flat/255, test_labels,\
                                   0.4, momentum_coefficient, 260, 38)
```

The below are the error metrics of the two separately trained networks on top each other in order to observe them better. The accuracy of the network with momentum on the test data has been found as 84%. The plots are given below.



Figures 13-14 - Training and Test Mean Squared Errors over Training Epochs with and without Momentum

Ayhan Okuyan
21601531



Figures 15-16 - Training and Test Accuracies over Training Epochs with and without Momentum

Here, we can easily observe that the movements of the errors are more stable in the momentum version of the network. In Figure 13, we observe that the MSE approaches zero faster. Between epochs 0 and 100 the rate of decrease is higher than the non-momentum network since the direction of the updates are directed more towards the minimum. Furthermore, the classification accuracy on the test data turned out to be better. Generally, we observe a more stable convergence.

Ayhan Okuyan
21601531

Question 3

This question asks us to implement a network that, given three words, the algorithm outputs the most suitable 4th word. The overall network structure is given below.

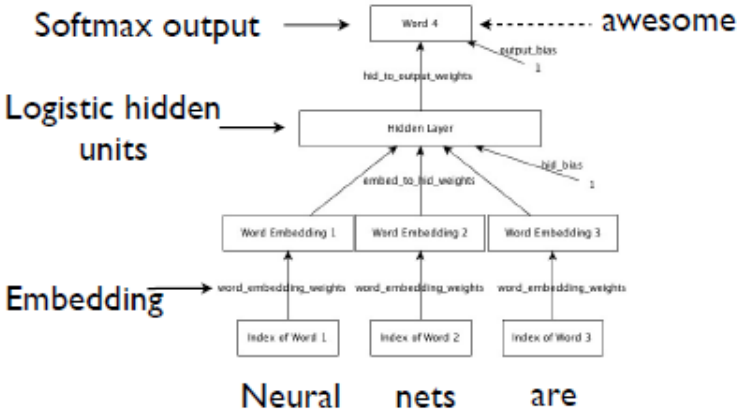


Figure 17 – The Neural Network Architecture for Question 3

The most challenging part of the assignment is the word embedding layer that we are required to implement. Assuming the inputs are one-hot encoded column vectors, The word embedding layer embedding matrix given in the dimensions of (dictionary size x D) is used to map that sparse one-hot input to a less sparse input space and each word is represented with a vector of length D.

$$Y_{WE} = X_{one-hot} * R$$

Furthermore, we are required to use sigmoid activations in the other hidden layer and SoftMax activation for the output layer activation, which are given below.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{p} e^{z_j}}, where\ p = \#neurons\ in\ the\ layer$$

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

In the applied task, p constant is given as 250. The hidden layer with sigmoid activation is required to have P neurons and the same embedding layer to be used in all of the word inputs.

Part A

This part of the question asks us to implement this network using a set of given but updateable parameters for better performance. The given parameters are as follows.

- $\eta = 0.15$
- $\alpha = 0.85$
- Mini Batch Size = 200
- Weight Standard Deviations = 0.01
- (D,P) = {(32,256),(16,128),(8,64)}

As mentioned, the most challenging part of the assignment is to find an applicable procedure for the implementation of the word embedding layer. There are two options that can be followed in the implementation process.

1- Inputting the samples and taking the sum of the transformed vectors and summing them to input them to the network in the dimensions (batchSizexD)

2- Inputting the samples and concatenating the outputs for separate results column-wise and inputting it to the network in the dimensions (batchSizex3D)

The option that I have chosen to implement is the second one since, summing the inputs removes the sentence structure from the algorithm meaning that the inputs “Ayhan goes to”, becomes the same as “To goes Ayhan” and I believe this was not an admissible option. Hence, I have separately inputted the words and concatenated the D-dimensioned vector outputs.

For this question, I have altered the neural network architecture that I have implemented in the previous question. I will be explaining the differences class by class as done with the Question 2 Part A.

Layer Class

In order to protect the structural form, I have thought of the Embedding Layer as one form of Layer with a linear activation (f(x)=x) since there is no form of activation mentioned. Furthermore, I have added an activation parameter to the input in order to apply several activations with only one class. The code for the constructor is given below.

```
class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons,
inputDim)
            self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
            self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
        elif self.activation == 'we':
            self.dictSize = numNeurons
            self.D = inputDim
            self.weights = np.random.normal(mean, std, dictSize*D).reshape((dictSize,D))


        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevUpdate = 0
```

Here, we can see that we have an if statement that separates what will be constructed according to the activation, for standard neuron layers, we have constructed the weight matrices and for the embedding layer, we have constructed the requested R matrix. inputDim and numNeurons were parameters that were used in the previous section, however, here, we have used them to acquire the dictionary size and D constant from the user to make the algorithm more generic.

After construction, we have changed the activation and the derivative function according to the activation functions that we were required to implement. The code is below.

```
    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'we'):
            return x
```

We have used this function in the activate function in order to update them with the layers’ inputs and weights which are given below.

```
    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*-1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        elif self.activation == 'we':
            layerOut = np.zeros((x.shape[0],x.shape[1], self.D))
            for m in range(layerOut.shape[0]):
                layerOut[m,:,:] = self.actFcn(np.matmul(x[m,:,:], self.weights))
            layerOut = layerOut.reshape((layerOut.shape[0], layerOut.shape[1] * layerOut.shape
[2]))
            self.lastActiv = layerOut.T
        return self.lastActiv
```

Then, we have followed the same procedure for the derivative function.

```
    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'we'):
            return np.ones(x.shape)
```

When backpropagating for the weight embedding layer, we have returned a ones matrix with the same shape as the input since the derivative of f(x)=x is 1. Derivative of SoftMax function is not given here the derivative will not be calculated directly which will be explained in the backProp function.

$$\frac{\partial(\sigma(x))}{x} = \sigma(x)(1-\sigma(x))$$

We will be using the same approach that we have used in Question Part A since the derivative of the function can be written in terms of the function itself. After completing the layer, we move onto the class structure in order to change the needed sections.

WordNet Class

The most important change has been implemented on the backProp, which is given below.

```
def backProp(self, inp, out, lrnRate, momCoeff, batchSize):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.delta = out - net_out
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inp
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*-1]]
            if(lyr.activation == 'sigmoid' or lyr.activation == 'softmax'):
                update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
                lyr.weightsE += update + momCoeff * lyr.prevUpdate
            elif(lyr.activation == 'we'):
                delta3d = lyr.delta.reshape((3,batchSize,lyr.D))
                inputToUse = np.transpose(inputToUse, (1,2,0))
                update = np.zeros((inputToUse.shape[1], delta3d.shape[2]))
```

```
                for i in range(delta3d.shape[0]):
                    update += lrnRate * np.matmul(inputToUse[i,:,:], delta3d[i,:,:])
                update = update/batchSize
                lyr.weights += update + momCoeff * lyr.prevUpdate
            lyr.prevUpdate = update
```

The changed parts are highlighted in red.

In this assignment, we have implemented the loss as the cross-entropy error loss, which Is given below.

$$\xi(\hat{Y}, Y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}\hat{y}_{ic}\log(y_{ic}) = -\frac{1}{N}\sum \hat{Y} * \log(Y^T)$$

The reason for that is, when the output layer is SoftMax and the error is cross-entropy, the delta of the output layer emerges as follows.

$$\delta_{output} = Y - \hat{Y}$$

Furthermore, when back propagating for the embedding layer, I needed to divide the update, back to the three-dimensional matrix form and averaged the divided matrices so that the update would be with the same dimensions as the R matrix. For the train, we have only altered the loss according to the description given above as

```
crossErr = - np.sum(np.log(valOutput) * outTest.T)/valOutput.shape[1]
```

The Optimizations and Results

After many trials and trying many parameters, I have configured the parameters as given below.

Hyper-Parameter	Selected Value
D	32
P	256
Weights Std	0.2
Learning Rate	0.35
Batch Size	250
Momentum Coefficient	0.85
Epoch Number	50
Beta of the Sigmoid Activation	2

Table 4 – Optimized Parameters for the

Here is the written code segment to create and run the training.

```
P = 256
D = 32
dictSize = 250

nn = WordNet()
nn.addLayer(Layer(D, dictSize, 'we', 0.2))
nn.addLayer(Layer(3*D, P, 'sigmoid', 0.2))
nn.addLayer(Layer(P, dictSize, 'softmax', 0.2))

learnRate = 0.35
momCoeff = 0.85
batchSize = 250
epoch = 50

val_inp_1H = mat1H(val_data, dictSize)
val_labels_1H = mat1H2(val_labels, dictSize)

errors = nn.train(train_data, train_labels, val_inp_1H, val_labels_1H,\
```

```
                    learnRate, momCoeff,\
                    epoch, batchSize)
```

In the result of 50 epochs, the validation accuracy turns out to be 35.42% and the cross-validation error on the last epoch is found as 2.74. The graph of the cross-validation errors over each epoch is given below.

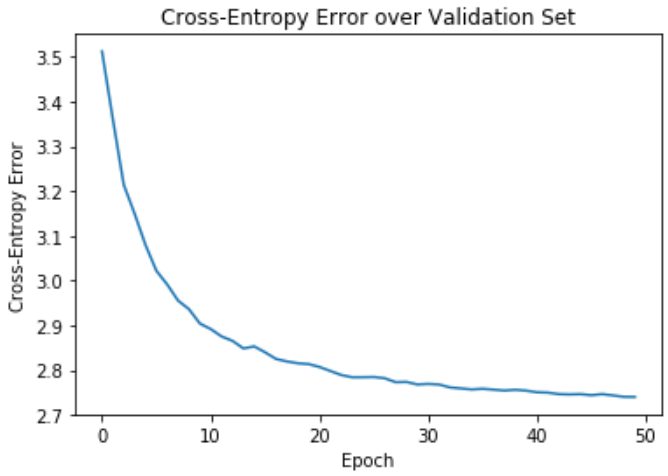


Figure 18 – The Cross-Validation Errors over Training

Here, we see a steady decrease in the error due to the high momentum coefficient.

Part B

This part asks us to take 5 random samples from the test data and forward the data through the trained network. Then, we should save the 10 word indexes with the highest probability and print them to see if they make sense. Hence, I have added a predictionTopK function to the WordNet class that serves this purpose. The top 10 predictions for the five words as given.

```
[1] long, time, ago
The Top-K predictions are: {i, at, right, ago, last, now, ,, ?, ., then, }
[2] do, nt, know
The Top-K predictions are: {any, when, how, ,, what, that, who, if, ., where, }
[3] see, who, 's
The Top-K predictions are: {not, on, there, going, right, out, at, up, here, in, }
[4] want, to, do
The Top-K predictions are: {what, more, is, ,, with, ?, that, this, ., it, }
[5] one, day, we
The Top-K predictions are: {could, will, know, have, do, did, can, were, want, are, }
```

Here, we see that the predictions are pretty sensible and the grammar even looks admissible, the sensibility is decreasing when moving down the list, which is also logical.

Appendix A - References

[1] “Gradient descent with constant learning rate for a quadratic function of multiple variables,” Calculus. [Online]. Available: https://calculus.subwiki.org/wiki/Gradient_descent_with_constant_learning_rate_for_a_quadratic_function_of_multiple_variables. [Accessed: 15-Nov-2019].

Appendix B – Codes

Question 2

```
#!/usr/bin/env python
# coding: utf-8
# The plots are intentionally separated with plt.show() commands for easy
evaluations
# In[1]:


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


# ## Part A

# In[2]:


filename = 'assign2_data1.h5'

with h5py.File(filename, 'r') as f:
    # List all groups
    print("Keys: %s" % f.keys())
    # Get the data
    test_img = f[list(f.keys())[0]].value
    test_labels = f[list(f.keys())[1]].value
    train_img = f[list(f.keys())[2]].value
    train_labels = f[list(f.keys())[3]].value
test_labels = test_labels.reshape((len(test_labels),1))
train_labels = train_labels.reshape((len(train_labels),1))


# In[3]:


class Layer:
    def __init__(self, inputDim, numNeurons, std, beta=1, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons

        self.beta = beta

        self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)

        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

    def activtanh(self, x):
        if(x.ndim == 1):
            x = x.reshape(x.shape[0],1)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(self.beta*np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def tanhDer(self, x):
        return self.beta*(1-(x**2))

    def __repr__(self):
        return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " +
```

```
str(self.numNeurons)


# In[4]:


class MLP:
    def __init__(self):
        self.layers = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            #print("input", inp.shape)
            out = lyr.activtanh(out)
            #print("output", inp.shape)
        return out

    #only for binary classificaiton with 1 output neurons
    def prediction(self, inp):
        out = self.forward(inp)
        out[out>=0] = 1
        out[out<0] = -1
        return out

    def backProp(self, inp, out, lrnRate, batchSize):

        net_out = self.forward(inp)
        #print('network out: ', net_out.shape)

        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]

            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.error = out - net_out
                '''print("out\n",out)
                print("net out\n", net_out)
                print("error = out - net_out\n", lyr.error)'''
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                #print("derMatrix = dertanh-lastActiv-\n", derMatrix)
                lyr.delta = derMatrix * lyr.error
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                #print("Error: \n", lyr.error)
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                #print("derMatrix \n", derMatrix)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            #print(inp)
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = np.r_[inp, [np.ones(numSamples)*-1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv,
[np.ones(numSamples)*-1]]
```

```
            lyr.weightsE += (lrnRate* np.matmul(lyr.delta, inputToUse.T))/batchSize
            '''print("Layer:", i)
            print("delta: \n", lyr.delta)
            print("inputTouse.T: \n", inputToUse.T)
            print("Product: \n", np.matmul(lyr.delta, inputToUse.T))
            print("Weights: \n", lyr.weightsE)'''


    def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize):
        mseList = []
        mceList = []
        mseTList = []
        mceTList = []

        for ep in range(epochNum):
            print('Epoch', ep)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]

            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                self.backProp(inp[batchSize*j:batchSize*j+batchSize].T,
out[batchSize*j:batchSize*j+batchSize].T, lrnRate, batchSize)

            mse = np.mean((out.T - self.forward(inp.T))**2, axis=1)
            mseList.append(mse)
            mce = np.sum(self.prediction(inp.T) == out.T)/len(out)*100
            mceList.append(mce)
            mseT = np.mean((outTest.T - self.forward(inpTest.T))**2, axis=1)
            mseTList.append(mseT)
            mceT = np.sum(self.prediction(inpTest.T) == outTest.T)/len(outTest)*100
            mceTList.append(mceT)
        return mseList, mceList, mseTList, mceTList

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr


# In[5]:


inputWH = train_img.shape[1]

train_img_flat = train_img.reshape(train_img.shape[0], inputWH**2)
test_img_flat = test_img.reshape(test_img.shape[0], inputWH**2)


# In[6]:


neuralNet = MLP()
neuralNet.addLayer(Layer(inputWH**2, 18, 0.02, 1))
neuralNet.addLayer(Layer(18, 1, 0.02, 1))

print(neuralNet)

train_labels[train_labels == 0] = -1

test_labels = test_labels.astype(int)
test_labels[test_labels == 0] = -1

mses, mces, mseTs, mceTs = neuralNet.train(train_img_flat/255, train_labels,
```

```
test_img_flat/255, test_labels, 0.35, 300, 38)
print('PART A IS DONE')

# In[7]:


print("Test Accuracy:", str(np.sum(neuralNet.prediction(test_img_flat.T/255).T ==
test_labels)/len(test_labels)*100) + "%")

plt.plot(mses)
plt.title('MSE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mces)
plt.title('MCE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()

plt.plot(mseTs)
plt.title('MSE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mceTs)
plt.title('MCE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()


# ## Part C

# In[8]:


neuralNetHigh = MLP()
neuralNetHigh.addLayer(Layer(inputWH**2, 800, 0.02, 1))
neuralNetHigh.addLayer(Layer(800, 1, 0.02, 1))
print(neuralNetHigh)

neuralNetLow = MLP()
neuralNetLow.addLayer(Layer(inputWH**2, 5, 0.02, 1))
neuralNetLow.addLayer(Layer(5, 1, 0.02, 1))
print(neuralNetLow)

msesH, mcesH, mseTsH, mceTsH = neuralNetHigh.train(train_img_flat/255,
train_labels, test_img_flat/255, test_labels, 0.35, 300, 38)
msesL, mcesL, mseTsL, mceTsL = neuralNetLow.train(train_img_flat/255, train_labels,
test_img_flat/255, test_labels, 0.35, 300, 38)
print('PART C IS DONE')

# In[9]:


plt.plot(mses)
plt.plot(msesH)
plt.plot(msesL)
plt.title('MSE Over Training')
plt.legend(['Optimal', 'High', 'Low'])
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mces)
```

```
plt.plot(mcesH)
plt.plot(mcesL)
plt.title('MCE Over Training')
plt.legend(['Optimal', 'High', 'Low'])
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()

plt.plot(mseTs)
plt.plot(mseTsH)
plt.plot(mseTsL)
plt.title('MSE Over Test')
plt.legend(['Optimal', 'High', 'Low'])
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mceTs)
plt.plot(mceTsH)
plt.plot(mceTsL)
plt.title('MCE Over Test')
plt.legend(['Optimal', 'High', 'Low'])
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()


# ## Part D

# In[10]:


neuralNetTwoHidden = MLP()
neuralNetTwoHidden.addLayer(Layer(inputWH**2, 450, 0.02, 1))
neuralNetTwoHidden.addLayer(Layer(450, 45, 0.02, 1))
neuralNetTwoHidden.addLayer(Layer(45, 1, 0.02, 1))

print(neuralNetTwoHidden)

mses2, mces2, mseTs2, mceTs2 = neuralNetTwoHidden.train(train_img_flat/255,
train_labels, test_img_flat/255, test_labels, 0.4, 260, 38)
print('PART D IS DONE')


# In[11]:


print("Test Accuracy:",
str(np.sum(neuralNetTwoHidden.prediction(test_img_flat.T/255).T ==
test_labels)/len(test_labels)*100) + "%")

plt.plot(mses2)
plt.title('MSE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mces2)
plt.title('MCE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()

plt.plot(mseTs2)
plt.title('MSE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()
```

```
plt.plot(mceTs2)
plt.title('MCE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()


# ## Part E

# In[12]:


class LayerwMom:
    def __init__(self, inputDim, numNeurons, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons

        self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)

        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevUpdate = 0

    def activtanh(self, x):
        if(x.ndim == 1):
            x = x.reshape(x.shape[0],1)
        numSamples = x.shape[1]
        tempInp = np.r_[x, [np.ones(numSamples)*-1]]
        self.lastActiv = np.tanh(0.1*np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def tanhDer(self, x):
        return 0.1*(1-(x**2))

    def __repr__(self):
        return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " +
str(self.numNeurons)

class MLPwMom:
    def __init__(self):
        self.layers = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activtanh(out)
        return out

    #only for binary classificaiton with 1 output neurons
    def prediction(self, inp):
        out = self.forward(inp)
        out[out>=0] = 1
        out[out<0] = -1
        return out

    def backProp(self, inp, out, lrnRate, momCoeff, batchSize):

        net_out = self.forward(inp)
```

```
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]

            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.error = out - net_out
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.tanhDer(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = np.r_[inp, [np.ones(numSamples)*-1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv,
[np.ones(numSamples)*-1]]
            update =  lrnRate * np.matmul(lyr.delta, inputToUse.T)
            lyr.weightsE += update + momCoeff * lyr.prevUpdate
            lyr.prevUpdate = update


    def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum,
batchSize):
        mseList = []
        mceList = []
        mseTList = []
        mceTList = []

        for ep in range(epochNum):
            print('Epoch', ep)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]

            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                self.backProp(inp[batchSize*j:batchSize*j+batchSize].T,
out[batchSize*j:batchSize*j+batchSize].T, lrnRate, momCoeff, batchSize)

            mse = np.mean((out.T - self.forward(inp.T))**2, axis=1)
            mseList.append(mse)
            mce = np.sum(self.prediction(inp.T) == out.T)/len(out)*100
            mceList.append(mce)
            mseT = np.mean((outTest.T - self.forward(inpTest.T))**2, axis=1)
            mseTList.append(mseT)
            mceT = np.sum(self.prediction(inpTest.T) == outTest.T)/len(outTest)*100
            mceTList.append(mceT)
        return mseList, mceList, mseTList, mceTList

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr
```

```
# In[13]:


neuralNetTwoHiddenM = MLPwMom()
neuralNetTwoHiddenM.addLayer(LayerwMom(inputWH**2, 450, 0.02))
neuralNetTwoHiddenM.addLayer(LayerwMom(450, 45, 0.02))
neuralNetTwoHiddenM.addLayer(LayerwMom(45, 1, 0.02))

print(neuralNetTwoHiddenM)

momentum_coefficient = 0.1

mses2M, mces2M, mseTs2M, mceTs2M = neuralNetTwoHiddenM.train(train_img_flat/255,
train_labels, test_img_flat/255, test_labels, 0.4, momentum_coefficient, 260, 38)
print('PART E IS DONE')


# In[14]:


print("Test Accuracy:",
str(np.sum(neuralNetTwoHiddenM.prediction(test_img_flat.T/255).T ==
test_labels)/len(test_labels)*100) + "%")

plt.plot(mses2)
plt.plot(mses2M)
plt.legend(['wo/Momentum','w/Momentum'])
plt.title('MSE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mces2)
plt.plot(mces2M)
plt.legend(['wo/Momentum','w/Momentum'])
plt.title('MCE Over Training')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()

plt.plot(mseTs2)
plt.plot(mseTs2M)
plt.legend(['wo/Momentum','w/Momentum'])
plt.title('MSE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.show()

plt.plot(mceTs2)
plt.plot(mceTs2M)
plt.legend(['wo/Momentum','w/Momentum'])
plt.title('MCE Over Test')
plt.xlabel('Epoch')
plt.ylabel('MCE')
plt.show()


# In[ ]:
```

Ayhan Okuyan
21601531

Question 3

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


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


# In[2]:


def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x-1] = 1
    return out

def mat1H(x, maxInd):
    out = np.zeros((x.shape[0], x.shape[1], maxInd))
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            out[i,j,:] = vector1H(x[i,j], maxInd)
    return out

def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out


# In[3]:


filename = 'assign2_data2.h5'

with h5py.File(filename, 'r') as f:
    # List all groups
    print("Keys: %s" % f.keys())
    # Get the data
    test_labels = f[list(f.keys())[0]].value
    test_data = f[list(f.keys())[1]].value
    train_labels = f[list(f.keys())[2]].value
    train_data = f[list(f.keys())[3]].value
    val_labels = f[list(f.keys())[4]].value
    val_data = f[list(f.keys())[5]].value
    wordDict = f[list(f.keys())[6]].value


# In[4]:


class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            self.weights = np.random.normal(mean,std,
inputDim*numNeurons).reshape(numNeurons, inputDim)
            self.biases = np.random.normal(mean,std,
numNeurons).reshape(numNeurons,1)
            self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
        elif self.activation == 'we':
```

```
            self.dictSize = numNeurons
            self.D = inputDim
            self.weights = np.random.normal(mean, std,
dictSize*D).reshape((dictSize,D))


        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevUpdate = 0


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

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*-1]]

            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        elif self.activation == 'we':
            layerOut = np.zeros((x.shape[0],x.shape[1], self.D))
            for m in range(layerOut.shape[0]):
                layerOut[m,:,:] = self.actFcn(np.matmul(x[m,:,:], self.weights))
            layerOut = layerOut.reshape((layerOut.shape[0], layerOut.shape[1] *
layerOut.shape[2]))
            self.lastActiv = layerOut.T
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return 2*(x*(1-x))
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'we'):
            return np.ones(x.shape)

    def __repr__(self):
        return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " +
str(self.numNeurons) + "\n Activation: " + self.activation


# In[5]:


class WordNet:
    def __init__(self):
        self.layers = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out
```

```
    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def predictionTopK(self, inp, k):
        out = self.forward(inp)
        return np.argpartition(out, -k, axis=0)[-k:]


    def backProp(self, inp, out, lrnRate, momCoeff, batchSize):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                lyr.delta = out - net_out
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            #write dynamic if later
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inp
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv,
[np.ones(numSamples)*-1]]
            if(lyr.activation == 'sigmoid' or lyr.activation == 'softmax'):
                update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
                lyr.weightsE += update + momCoeff * lyr.prevUpdate
            elif(lyr.activation == 'we'):
                delta3d = lyr.delta.reshape((3,batchSize,lyr.D))
                inputToUse = np.transpose(inputToUse, (1,2,0))
                update = np.zeros((inputToUse.shape[1], delta3d.shape[2]))
                for i in range(delta3d.shape[0]):
                    update += lrnRate * np.matmul(inputToUse[i,:,:],
delta3d[i,:,:])
                #mean the updates for each separate word
                #update = update/delta3d.shape[0]
                update = update/batchSize
                lyr.weights += update + momCoeff * lyr.prevUpdate
            lyr.prevUpdate = update

    def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum,
batchSize):
        cveList = []

        for ep in range(epochNum):
            print('Epoch', ep)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))
```

```
            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_inp_1H = mat1H(batch_inp, dictSize)
                batch_out_1H = mat1H2(batch_out, dictSize).T

                self.backProp(batch_inp_1H, batch_out_1H, lrnRate, momCoeff,
batchSize)

            valOutput = self.forward(inpTest)

            crossErr = - np.sum(np.log(valOutput) * outTest.T)/valOutput.shape[1]
            print('Cross-Entropy Error ', crossErr)
            cveList.append(crossErr)

            valAcc = np.sum(self.prediction(inpTest) == np.argmax(outTest.T,
axis=0))
            print('Correct: ', valAcc)
            print('Accuracy: ', valAcc/valOutput.shape[1])

        return cveList

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr


# In[6]:


P = 256
D = 32
dictSize = 250

nn = WordNet()
nn.addLayer(Layer(D, dictSize, 'we', 0.2))
nn.addLayer(Layer(3*D, P, 'sigmoid', 0.2))
nn.addLayer(Layer(P, dictSize, 'softmax', 0.2))

learnRate = 0.35
momCoeff = 0.85
batchSize = 250
epoch = 50

val_inp_1H = mat1H(val_data, dictSize)
val_labels_1H = mat1H2(val_labels, dictSize)

errors = nn.train(train_data, train_labels, val_inp_1H, val_labels_1H, learnRate,
momCoeff, epoch, batchSize)


# In[26]:


num_samples = 5

random_sample_indexes = np.random.permutation(len(test_data))[0:num_samples]

test_samples = test_data[random_sample_indexes]
test_outputs = test_labels[random_sample_indexes]

test_samples_1H = mat1H(test_samples, dictSize)
test_labels_1H = mat1H2(test_outputs, dictSize)

top10predictions = nn.predictionTopK(test_samples_1H, 10)
```

```
for i in range(num_samples):
    print('[' + str(i+1) + '] ' + str(wordDict[test_samples[i,0]-1].decode("utf-
8"))  + ', ' +           str(wordDict[test_samples[i,1]-1].decode("utf-8"))
+ ', ' + str(wordDict[test_samples[i,2]-1].decode("utf-8")))
    strin = 'The Top-K predictions are: {'
    for j in range(10):
        strin += (str(wordDict[top10predictions[j,i]].decode("utf-8"))) + ', '
    print(strin + '}')
```