EEE 443 – Neural Networks

Mini Project Report

Ata Berk Çakır - 21703127

List of Figures

1. Question 1

1.a. Image Preprocessing and Displaying Sample Images

In the first question we are required to an autoencoder neural network with a single hidden layer for unsupervised feature extraction from natural images. Different than a normal machine learning problem since our data is images, there are a few problems associated with image data include complexity, inaccuracy, and inadequacy. Therefore, before building a such computer vision task, it is essential to preprocess the data. There is a function called "*normalize ()"* created in the code to do the following tasks . Our original data contains RGB images. Using RGB images for this task is not the best option, thus using the formula given below first RGB images turned into gray scale images.

$$Y = 0.2126 * R + 0.7152 * G + 0.0722 * B$$

Then respectively, mean pixel intensity removed from each image itself with using formula [-3*std,3*std] where std represents the standard deviation. To prevent saturation data is scaled into the range [0.1,0.9]. To do so max-min scaler formula given below where $x_{min} = 0.1$ and $x_{max} = 0.9$

$$x^{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

To display random 200 images first "*random_index()"* function created that returns 200 random index in a list using random.randint method. After picking random sample using *"show_images()"* and "*show_gray_images()"* functions, the following subplots contain 200 random RGB and grayscale images are plotted.

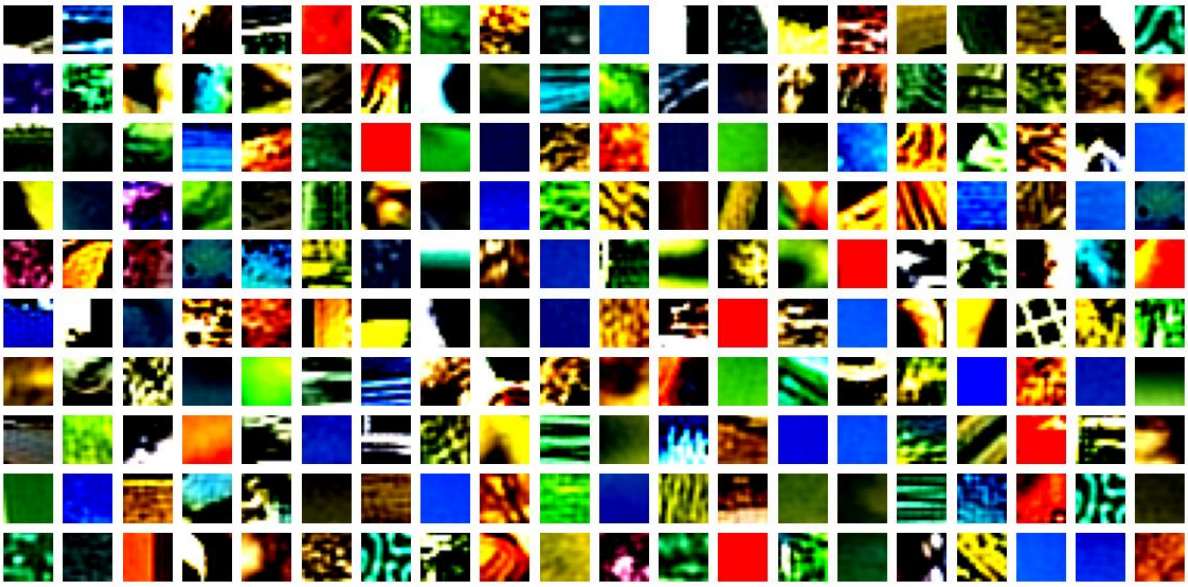

Figure 1 200 Randomly Selected RGB Images

Figure 2 200 Randomly Selected Normalized Gray Scale Images

Figure 2 shows the normalized and gray scaled version of Figure 1. By just looking those figures we can state that in the normalized version the shapes and lines on the images much more apparent and precise compared to the RGB version. Since we are done with the image preprocessing part and get the desired normalized and gray scaled data we can move on to the next part.

1.b. Neural Network Implementation

Before moving on to the implementation of the NN, firstly *"initialize weights ()"* and *"initialize_params()"* functions are created to make proper initializations asked in the project description. *"initialize_weights(Lin, Lhid*)" takes number of neurons in the input layer and number of neurons in the hidden layer. Since Lin == Lout taking Lout as input is not necessary. Function returns W1,W2,b1,b2 weights and biases in a list called "we". weights and biases created using np.random.uniform method in the range of [-w0,w0]. w0 is defined as following.

$$wo = \sqrt{\frac{6}{L_{pre} + L_{post}}}$$

After initializing weights, biases and parameters I crated a function called "*forward (data, we)"* which takes data and initialized weights list, makes forward propagation task and returns output of input layer as A1 and hidden layer as output separately. In the forward propagation process, data is multiplied with weights than bias term is added and finally result taken into sigmoid function. This process is repeated twice since we have 2 layered NN structure. Sigmoid function used for forward propagation defined as following.

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

The trickiest part of this question is to implement "aeCost()" which returns total cost and calculated gradients for each weight and bias separately. The cost function to minimize for this question is the following.

$$J_{ae} = \frac{1}{2N}\sum_{i=1}^{N}\|d(m) - o(m)\|^2 + \frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2 + \sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}\left(W_{b,c}^{(2)}\right)^2\right] + \beta\sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b)$$

Figure 3 Cost Function

$\widehat{p_b}$ is the average activation of hidden unit h across training samples which can be calculated by following.

$$\widehat{p_b} = \frac{1}{N}\sum_{n=0}^{N} h_n$$

In the aeCost function total cost calculated by separately for the easiness of the calculated and then added to create one single total cost.

J = average_squared_error + tykhonov + kl_divergence

The partial derivatives calculated by using wolframalpha since cost function is long and hard to taking derivatives by hand. In order to update parameters according to learning rate *"backward ()"* function is created which returns updated weights and cost. Finally, "train ()" function performs full batch gradient on the NN structure according to epoch number to minimize the cost.

After many trials the optimized parameters found given in the following table. Also, loss vs epoch graph of model that trained with those parameters provided.

Lin	256
Lout	256
Lhid	64
Lambda	5e-4
Beta	0.01
rho	0.03
Learning rate	0.7

Figure 4 Hyper parameter table

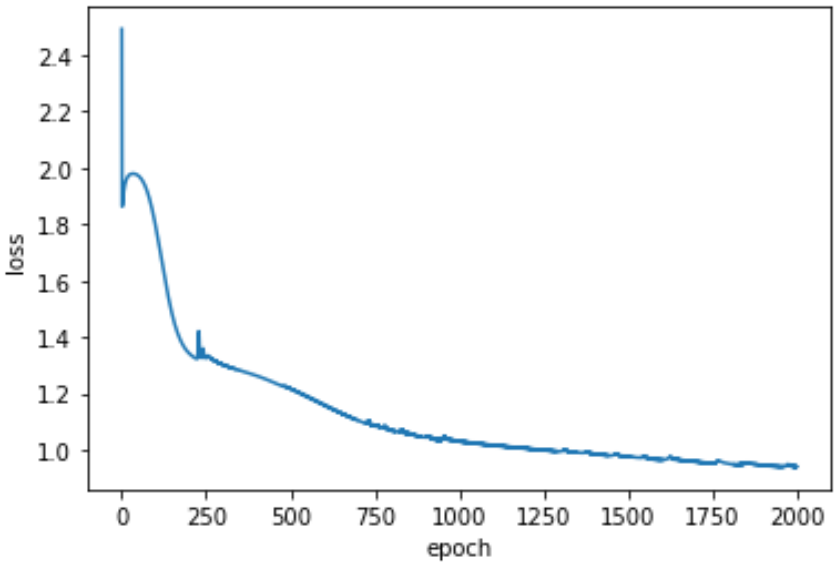

Figure 5 Loss vs Epoch Graph

1.c. Display Trained Network Weights

In this part we are required to display learned weights according to the parameter setup given in table above part 1.b. The image of weights for each hidden layer neuron given below as subplot. For this task *"plot_weights(we)"* function in code is used. Figure below we can see that the weight matrices are similar to the natural images provided in the dataset. These images are not typical of natural images, in the sense that they do not visualize a natural image, but they are portions of a natural image. We can observe that the data contains a variety of patterns, some of which appear several times. While some of them are rather distinct, we can see that some are quite similar. This might indicate that there are sufficient hidden layer units present to learn the data or provide more specific information about the data distribution, implying that some patterns are more frequent in the data than others.

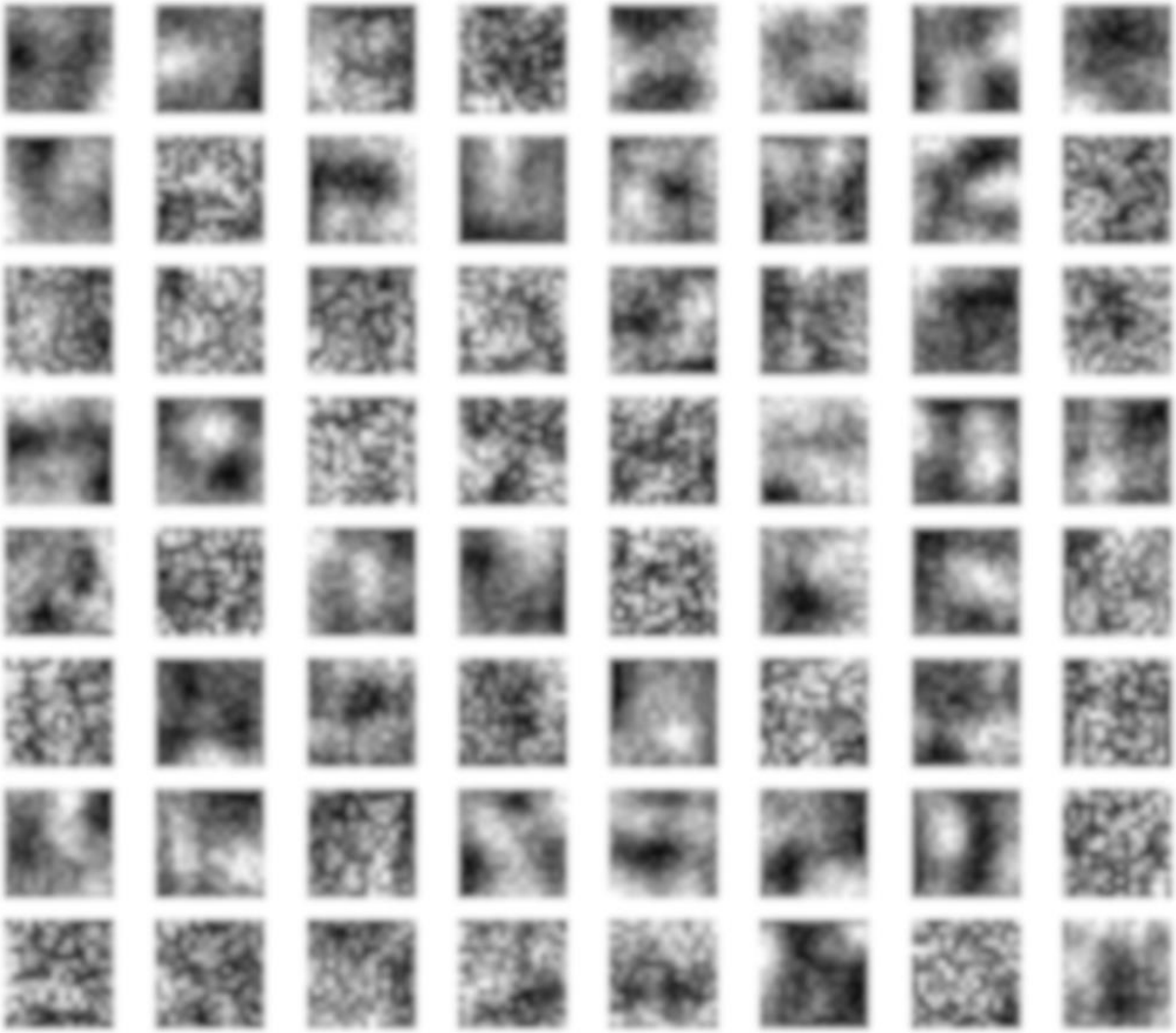

Figure 6 First Layer of Connection Weights as Separate Images

1.d. Trials with Different L_{hid} and λ

In this part we are asked the retrain model network with 3 different $L_{hid} \in [10,100]$ and $\lambda \in [0, 10^{-3}]$ while keeping beta and rho parameters set at part b fixed. For that purpose, I chose L_{hid} values as 16, 49 ,81 and λ values as 0, 10^{-6}, 10^{-3}. In the next page weights as separate images will be provided for 9 different combinations chosen. In the below I will provide my discussion about different trials as bullet points.

- Different characteristics are retrieved when the hidden unit size (L_{hid}) increases. This is beneficial because diverse representations of objects are desired. However, this situation also causes extraction of some redundant features too as seen in the figure 7 when the L_{hid} increases, some images start to become very similar.
- Moreover, when L_{hid} increases learning process smoothens and become faster. As a result, high L_{hid} values return lower costs after training. In the two plots given below proves this fact.

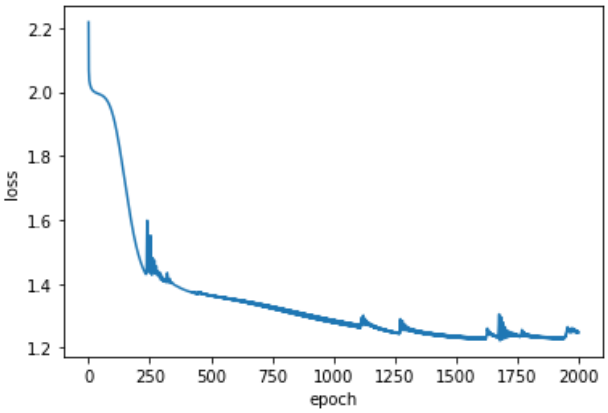

Figure 7 Epoch vs Loss (L_hid=16 and alpha = 1e-3)

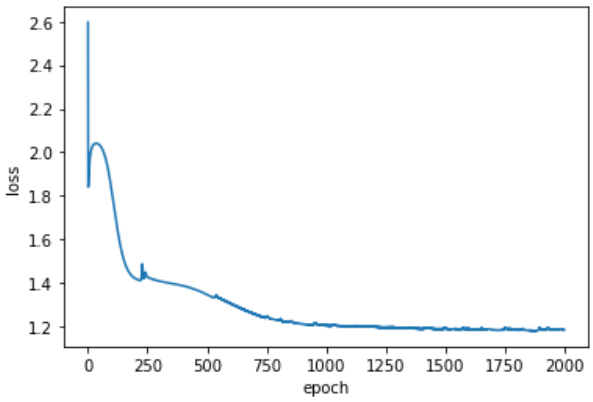

Figure 8 Epoch vs Loss (L_hid=81 and alpha = 1e-3)

- The value of the regularization hyperparameter changes the output in a variety of ways. Regularization minimizes the model's variance while maintaining its bias. As a result, the tuning parameter λ regulates the effect on bias and variance. As the value of λ decreases, the value of coefficients decreases, and therefore the variance decreases. This increase λ in is useful up to a degree because it merely reduces variance (thus preventing overfitting) without sacrificing any significant characteristics in the data. However, at a certain value, the model begins to lose crucial characteristics, resulting rise in bias and underfitting. Because of the reasons stated, the value of λ should be carefully chosen.
    - $\lambda = 0$ value resulting overfitting which is expected there is no regularization
    - $\lambda = 10^{-6}$ value gives very similar results to 0. Model still overfits the data when we look the final loss.
    - $\lambda = 10^{-3}$ value seems to give the best result. When we looked the figures, lines are more precise and show better quality of features.

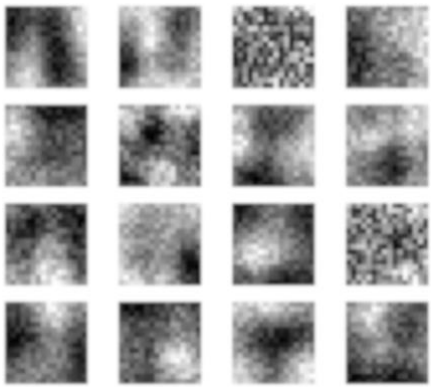

Figure 9 Lhid = 16 - alpha = 0

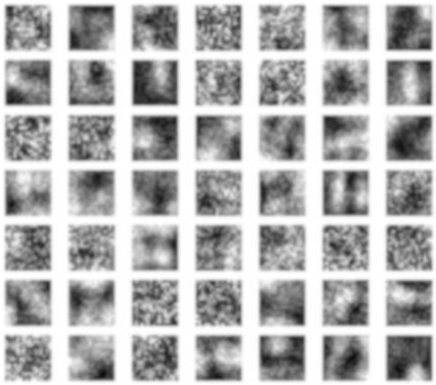

Figure 10 Lhid = 49 - alpha = 0

Figure 11 Lhid = 81 - alpha = 0

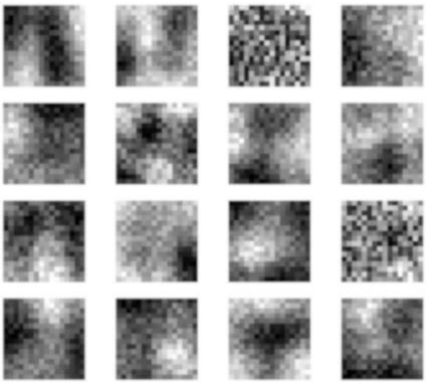

Figure 12 Lhid = 16 - alpha = 1e-6

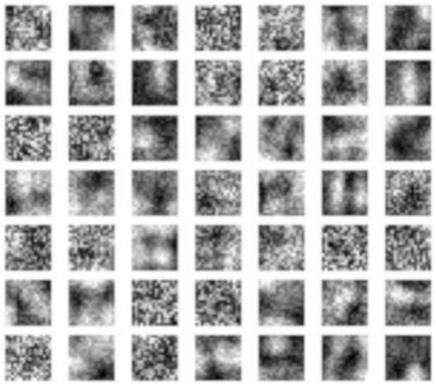

Figure 13 Lhid = 49 - alpha = 1e-6

Figure 14 Lhid = 81 - alpha = 1e-6

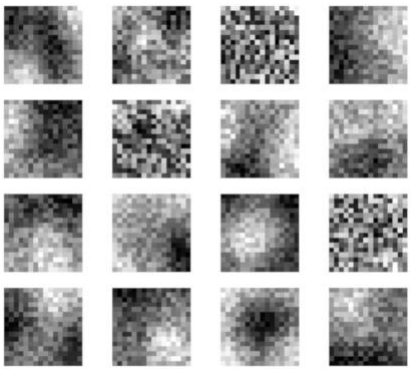

Figure 15 Lhid = 16 - alpha = 1e-3

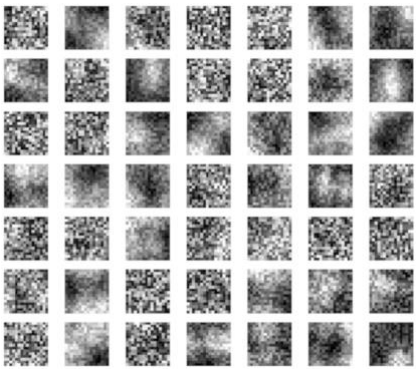

Figure 16 Lhid = 49 - alpha = 1e-3.

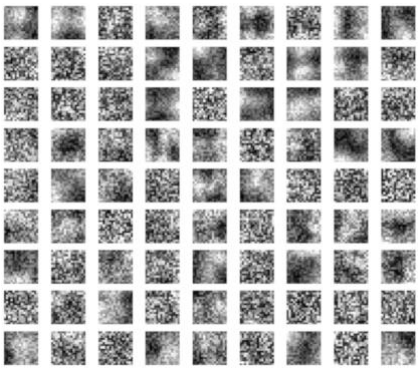

Figure 17 Lhid = 81 - alpha = 1e-3

2. Question 2

2.a. NN Structure

In the second question we are required to create a Neural network architecture to predict the fourth word in sequence given the preceding trigram. N-grams is a standard approach to a language modelling. The main task is to compute a probability of a sentence W with the given formula above.

$$P(W) = \prod_i P(w_i \,|w_1 \dots \dots w_{i-1})$$

In our case this formula can simplified to trigrams.

$$P(W) = \prod_i P(w_i \,|w_{i-2} \dots \dots w_{i-1})$$

In theory a good model makes the following probabilistic calculation

$$P(\text{"this is sentence"}) > P(\text{sentence is this"}) > \text{P("}sdkjsdsjk\text{"})$$
$$P(\text{"this is sentence"}) = P(\text{this}) \times P(is|this) \times P(sentence|this, is)$$

The overall structure of the model is given below

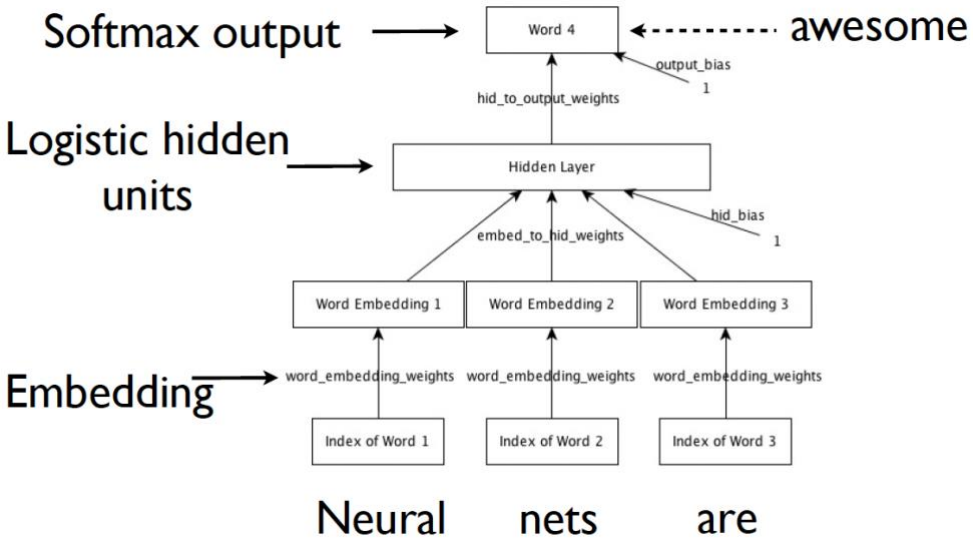

Figure 18 NN Structure

Before moving to creating the network, it is important to decide embedding part of this structure. After some research on this topic, I found that there exist two different accepted method for this task.

- First approach is to input the samples and concatenating the outputs for distinct findings column-by-column before feeding it into the network.
- Second approach is to input the samples and summing the converted vectors before sending them to the network.

I chose to move on with the second approach for the simplicity of the network structure.

Since the network cannot understand language, it needs numerical numbers to help it process the input. We must transform categorical data to numbers before applying any sort of algorithm on the data. To do this I used one hot encoding method that transforms categorical variables to binary vectors. In our case since we have 250 different words in the dictionary length of each word vector is equal to 250, all columns are sparse expect the column which is equal to index of that word coded as 1. The example given below gives better understanding how one hot encoding works.

Example:

```
vocabulary = (Monday, Tuesday, is, a, today)
Monday   = [1 0 0 0 0]
Tuesday  = [0 1 0 0 0]
is       = [0 0 1 0 0]
a        = [0 0 0 1 0]
today    = [0 0 0 0 1]
```

Hence our original summed input data in shape (N,3) becomes (N,750).

The structure of the networks is the following:

- The network consists of 2 hidden layers hence it can be sad that has 3 layered structure. Input layer has of 750 then hidden layers have size of D and P respectively and finally output has a size of 250 which is a vector consists of only one column contains “1”. When decoded this column will give us the index of the predicted word.

- Weights are initialized using “initialize_weights(D,P,data,mean=0,std=0.01)” function in the code uses random Gaussian variables of std =0.01 while initializing. W0, W1, W2, b1, b2 have the following shapes respectively (750,D), (D,P), (P,250), (batch_size,P), (batch_size,250).

- Second hidden layer has sigmoid function as activation function and output layer uses softmax as a activation function. Also, as a loss function cross entropy loss which measures the performance of a classification model whose output is a probability value between 0 and 1 is used.

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

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$crossentropy = -\sum_{i=0}^{M} y_i \log\left(y_{pred_i}\right)$$

- The rest of the implementation is similar to the any multi-hidden layer network and similar functions used in Question 1 is used such as “forward(), calculate_cost(), backward()”.

The network trained with 3 different D and P values which are (8,64) - (16,128) - (32,256) the performance of the network is measured using the test and validation set. Although the difference is very low higher number of neurons gives better performance.

D, P	Train Loss	Validation Loss
(32,256)	5.28	5.30
(16,128)	5.297	5.312
(8,64)	5.301	5.336

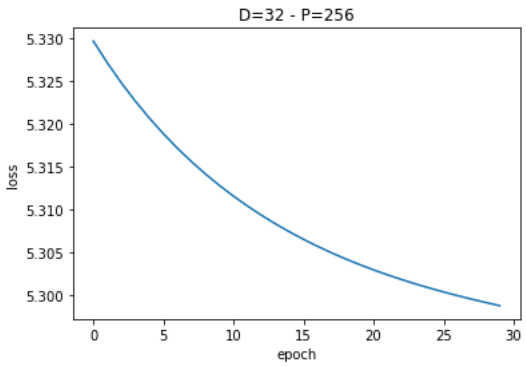

Figure 19 epoch vs loss (D=32 - P=256)

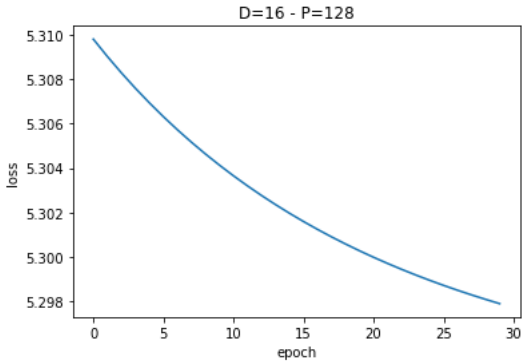

Figure 20 epoch vs loss (D=16 - P=128)

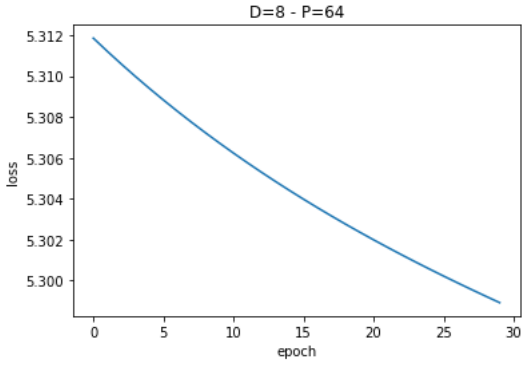

Figure 21 epoch vs loss (D=8 - P=64)

2.b. Predictions

In this part of the question pick some sample trigrams from the test data, and generate predictions for the fourth word via the trained neural network. Samples are picked via "pick_sample(data,label,sample_size)" function in the code. Using "predict(words,output)" and "print_preds(random_sample,test_label,pred_rows,words)" functions the following output is taken. Predictions made by using the network where D = 32 and P = 256.

```
sample trigram: team few day  ----> label: them
Top 10 predictions:  ['back', 'big', 'group', 'them', 'called', 'second', 'director', 'west', 'among', 'us']
sample trigram: me , day  ----> label: by
Top 10 predictions:  ['today', 'well', 'day', 'government', 'american', 'three', ' by ', 'political', 'percent', 'between']
sample trigram: many including way  ----> label: including
Top 10 predictions:  ['season', 'here', 'times', 'few', 'states', 'on', 'by', 'part', 'put', 'every']
sample trigram: well them day  ----> label: street
Top 10 predictions:  ['.', 'do', 'year', 'his', 'you', 'to', '?', 'the', 'work', 'university']
sample trigram: days we left  ----> label: says
Top 10 predictions:  ['.', '?', 'do', 'to', 'says', 'it', 'you', ',', 'that', 'nt']
```

Figure 22Predicted Outputs

Predictions seems not much accurate however we are also able to predict some labels true not in first candidate but in different levels of candidates. Let's discuss the reasons behind this situation. There are countless alternatives for the meaningful fourth word in a three-word sequence. I think network struggles which one is the most proper one. For certain phrases, the labels teach the word just one sensible result, and when another of the same general phrase is given, the network fails to guess the proper label, instead predicting the one it learnt. When we look at the first prediction we were able to predict the correct label in 4^{th} candidate. Also for second word 7^{th} candidate and fifth word 5^{th} candidate were the correct labels. At the beginning of this question we talked about 2 different approaches to deal with the embedding part. I believe choosing the first method might give better predictions because summing the inputs eliminates the sentence structure from the algorithm, resulting in "I have to" becoming "To have I" which actually are not same.

Appendix

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May  3 11:53:37 2022

@author: ata berk çakır 21703127
"""

import h5py
import random
import numpy as np
from matplotlib.pyplot import imshow, show, subplot, figure, axis, plot, xlabel,ylabel,title,savefig

def q1():

  ######################################## Part A ###################################

  #read data from .h5 file
  filename = "data1.h5"

  with h5py.File(filename, "r") as f:
    # List all groups
    print("Keys: %s" % f.keys())
    a_group_key = list(f.keys())[0]

    # Get the data
    #data = list(f[a_group_key])
    data = np.array(f[a_group_key])

  def normalize (data):
    #turn rgb to graysclae
    data_gray = np.array(data)[:,0,:,:]*0.2126 + np.array(data)[:,1,:,:]*0.7152 + np.array(data)[:,2,:,:]*0.0722
    mean = np.mean(data_gray,axis=(1,2)) #find mean

    for i in range (len(mean)):
      data_gray[i,:,:] -= mean[i] #remove the mean pixel intensity of each image from itself
```

```
    std = np.std(data_gray)  # find std
    data_gray = np.clip(data_gray, - 3 * std, 3 * std)  # clip -+3 std
    #normalize and map to 0.1 - 0.9
    normalized = data_gray*(0.90 - 0.10)/(2*np.max(data_gray))
    normalized += (0.10 - np.min(normalized))
    return normalized

  def random_index():
    random_index = []
    for i in range(200):
      index =random.randint(0,10239)
      random_index.append(index)
    return random_index


  def show_images(data,random_pics):
    figure(figsize=(20,10))
    j=0
    for i in random_pics:
      j+=1
      img = (data[i]).T
      subplot(10, 20, j)
      imshow(img)
      axis("off")
    savefig("fig1")

  def show_gray_images(data,random_pics):
    figure(figsize=(20,10))
    j=0
    for i in random_pics:
      j+=1
      img = (data[i]).T
      subplot(10, 20, j)
      imshow(img,cmap="gray")
      axis("off")
    savefig("fig2")

  ###################################  PART  B
###################################

  def w0 (Lpre,Lpost):
```

```
    w0 = np.sqrt(6/(Lpre + Lpost))
    return w0

def initialize_weights(Lin, Lhid):

    Lout = Lin
    np.random.seed(42)

    W1 = np.random.uniform(-w0(Lin,Lhid),w0(Lin,Lhid), size = (Lin,Lhid))
    W2 = np.random.uniform(-w0(Lhid,Lout),w0(Lhid,Lout), size = (Lhid,Lout))
    b1 = np.random.uniform(-w0(1,Lhid),w0(1,Lhid), size = (1,Lhid))
    b2 = np.random.uniform(-w0(1,Lout),w0(1,Lout), size = (1,Lout))

    we = [W1,W2,b1,b2]

    return we

def initialize_params(Lin,Lhid,lmbd, beta, rho):
    params = [Lin,Lhid,lmbd,beta,rho]

    return params

def sigmoid(x):
    y = 1 / (1 + np.exp(-x))
    return y

def sigmoid_backward(x):
    d_sig = x*(1-x)
    return d_sig

def forward(data, we):

    w1,w2,b1,b2 = we

    z1 = np.dot(data,w1) + b1 #first layer linear forward
    A1 = sigmoid(z1) #first layer activation
    z2 =  np.dot(A1,w2) + b2 #output linear forward
    output = sigmoid(z2) #output layer activation

    return A1, output
```

```
def aeCost(we, data, params):

    Lin,Lhid,lmbd,beta,rho = params
    w1,w2,b1,b2 = we
    N = len(data)

    A1, output = forward(data, we)
    mean = np.mean(A1, axis=0)

    #calculate cost
    average_squared_error = (1/(2*N))*np.sum((data-output)**2)
    tykhonov = (lmbd/2)*(np.sum(w1**2) + np.sum(w2**2))
    kl_divergence = beta*np.sum((rho*np.log(mean/rho))+((1-rho)*np.log((1-mean)/(1-rho))))
    J = average_squared_error + tykhonov + kl_divergence
    d_hid = (np.dot(w2,(-(data-output)*sigmoid_backward(output)).T)+ (np.tile(beta*(-(rho/mean.T)+((1-rho)/(1-mean.T))), (10240,1)).T)) * sigmoid_backward(A1).T

    d_w1 = (1/N)*(np.dot(data.T,d_hid.T) + lmbd*w1)
    d_w2 = (1/N)*(np.dot((-(data-output)*sigmoid_backward(output)).T,A1).T + lmbd*w2)
    d_b1 = np.mean(d_hid, axis=1)
    d_b2 = np.mean((-(data-output)*sigmoid_backward(output)), axis=0)

    Jgrad = [d_w1,d_w2,d_b1,d_b2]

    return J, Jgrad


def backward(data, lr_rate, we, params):
    #get gradients
    J, Jgrad = aeCost(we, data, params)
    #update weights
    we[0] -= lr_rate*Jgrad[0]
    we[1] -= lr_rate*Jgrad[1]
    we[2] -= lr_rate*Jgrad[2]
    we[3] -= lr_rate*Jgrad[3]
    return J, we

def train(data_gray,epoch,Lin,Lhid,lmbd, beta, rho,lr_rate):
    losses = []
    epochs = []
```

```
    data_flat = np.reshape(data_gray, (data_gray.shape[0],data_gray.shape[1]**2))
    we = initialize_weights(Lin,Lhid)
    params = initialize_params(Lin,Lhid,lmbd, beta, rho)


    for i in range (epoch):
        J, we = backward(data_flat, lr_rate, we, params)
        epochs.append(i)
        losses.append(J)
        print("Epoch: {} ----------> Loss: {} ".format(i+1,J))

    return we,losses,epochs

def plot_losses(losses,epochs):
    xlabel("epoch")
    ylabel("loss")
    plot(epochs,losses)


def plot_weights(we,name):
    w1,w2,b1,b2 = we
    figure(figsize=(18, 16))
    plot_shape = int(np.sqrt(w1.shape[1]))
    for i in range(w1.shape[1]):
        subplot(plot_shape,plot_shape,i+1)
        imshow(np.reshape(w1[:,i],(16,16)), cmap='gray')
        axis('off')
    savefig(name)
############################### PART A ############################
index = random_index()
show_images(data,index)
data_gray = normalize (data)
show_gray_images(data_gray,index)
############################### PART C ############################
we,losses,epochs = train(data_gray,100,256,64,5e-4, 0.01, 0.03,0.7)
"""plot_losses(losses,epochs)"""
plot_weights(we,"fig3")
############################### PART D ############################
############ Lhid = 16, alpha = 0 ############################
we,losses,epochs = train(data_gray,2000,256,16,0, 0.01, 0.03,0.7)
plot_weights(we,"fig4")
```

```
    ############ Lhid = 49, alpha = 0 ############################
    we,losses,epochs = train(data_gray,2000,256,49,0, 0.01, 0.03,0.7)
    plot_weights(we,"fig5")
    ############ Lhid = 81, alpha = 0 ############################
    we,losses,epochs = train(data_gray,2000,256,81,0, 0.01, 0.03,0.7)
    plot_weights(we,"fig6")
    ############ Lhid = 16, alpha = 1e-3 ############################
    we,losses,epochs = train(data_gray,2000,256,16,1e-3, 0.01, 0.03,0.7)
    plot_weights(we,"fig7")
    plot_losses(losses,epochs)
    ############ Lhid = 49, alpha = 1e-3 ############################
    we,losses,epochs = train(data_gray,2000,256,49,1e-3, 0.01, 0.03,0.7)
    plot_weights(we,"fig8")
    ############ Lhid = 81, alpha = 1e-3 ############################
    we,losses,epochs = train(data_gray,2000,256,81,1e-3, 0.01, 0.03,0.7)
    plot_weights(we,"fig9")
    plot_losses(losses,epochs)
    ############ Lhid = 16, alpha = 1e-6 ############################
    we,losses,epochs = train(data_gray,1000,256,16,1e-6, 0.01, 0.03,0.7)
    plot_weights(we,"fig10")
    ############ Lhid = 49, alpha = 1e-6 ############################
    we,losses,epochs = train(data_gray,1000,256,49,1e-6, 0.01, 0.03,0.7)
    plot_weights(we,"fig11")
    ############ Lhid = 81, alpha = 1e-6 ############################
    we,losses,epochs = train(data_gray,1000,256,81,1e-6, 0.01, 0.03,0.7)
    plot_weights(we,"fig12")



def q2():
    filename = "data2.h5"
    with h5py.File(filename, "r") as f:
        # List all groups
        print("Keys: %s" % f.keys())
        y_test = f[list(f.keys())[0]][:]
        x_test = f[list(f.keys())[1]][:]
        y_train = f[list(f.keys())[2]][:]
        x_train = f[list(f.keys())[3]][:]
        y_val = f[list(f.keys())[4]][:]
        x_val = f[list(f.keys())[5]][:]
        words = f[list(f.keys())[6]][:]
```

```
def word_vector(x, maxInd = 250):
    out = np.zeros(maxInd)
    out[x-1] = 1
    return out

def input_vector(data):
    encoded_data = []
    for row in data:
        word_1 = word_vector(row[0])
        word_2 = word_vector(row[1])
        word_3 = word_vector(row[2])
        row = np.concatenate((word_1,word_2,word_3))
        row = row.reshape(1,len(row))
        encoded_data.append(row)
    return encoded_data

def output_vector(data):
    encoded_data = []
    for row in data:
        word = word_vector(row)
        encoded_data.append(word)
    return encoded_data

def initialize_weights(D,P,data,mean=0,std=0.01):
    N = 200
    W0 = np.random.normal(mean,std, 750*D).reshape(750, D)
    W1 = np.random.normal(mean,std, D*P).reshape(D,P)
    W2 = np.random.normal(mean,std, P*250).reshape(P,250)
    b1 = np.random.normal(mean,std, N*P).reshape(N,P)
    b2 = np.random.normal(mean,std, N*250).reshape(N,250)

    we = [W0,W1,W2,b1,b2]

    return we

def sigmoid(x):
    y = 1 / (1 + np.exp(-x))
    return y

def sigmoid_backward(x):
```

```
    d_sig = x*(1-x)
    return d_sig

def softmax(x):
    expx = np.exp(x - np.max(x))
    y = expx/np.sum(expx, axis=0)
    return y

def cross_entrophy(y,y_pred):
    return (np.sum(- y * np.log(y_pred))/ y.shape[0])

def forward(data, we):

    w0,w1,w2,b1,b2 = we

    z0 = np.dot(data,w0) #first layer linear forward
    A0 = z0 #first layer without activation
    z1 = np.dot(A0,w1) + b1 #second layer linear forward
    A1 = sigmoid(z1) #second layer activation
    z2 =  np.dot(A1,w2) + b2 #output linear forward
    output = softmax(z2) #output layer activation

    return A0,A1,output

def calculate_cost(data,y,we):
    N = data.shape[0]
    W0,W1,W2,b1,b2 = we
    A0,A1,output = forward(data, we)
    d_sig = sigmoid_backward(A1)
    #calculate cost
    cost = cross_entrophy(y,output)
    #calculate gradients
    d_w0 = np.dot(data.T,((np.dot(((np.dot((y-output),W2.T))*d_sig),W1.T))*A0))
    d_w1 = np.dot(A0.T,(np.dot((y-output),W2.T)*d_sig))
    d_w2 = np.dot(A1.T,(y-output))
    d_b1 = np.dot((y-output),W2.T)*d_sig
    d_b2 = y-output

    grads = [d_w0,d_w1,d_w2,d_b1,d_b2]

    return cost, grads
```

```
    def backward(data,y,lr_rate,momentum, we, old_grads):
        #get gradients
        cost, grads = calculate_cost(data,y,we)
        #update weights
        we[0] -= lr_rate*grads[0]+old_grads[0]*momentum
        we[1] -= lr_rate*grads[1]+old_grads[1]*momentum
        we[2] -= lr_rate*grads[2]+old_grads[2]*momentum
        we[3] -= lr_rate*grads[3]+old_grads[3]*momentum
        we[4] -= lr_rate*grads[4]+old_grads[4]*momentum

        return cost, grads, we

    def train(x_train,y_train,D,P,epoch,num_batch,lr_rate,momentum):
        costs = []
        epochs = []
        data = input_vector(x_train)
        data = np.squeeze(data,axis=1)
        label = output_vector(y_train)
        label = np.array(label)
        we = initialize_weights(D,P,data,mean=0,std=0.01)
        momentum = 0.0000085
        cost, old_grads = calculate_cost(data[:200],label[0:200],we)
        lr_rate = 0.0000015
        for i in range (epoch):
            epochs.append(i)
            for j in range (num_batch):
                data_batch = data[200*j:200*j+200]
                label_batch = label[200*j:200*j+200]
                cost,  grads,we  =  backward(data_batch,label_batch,
lr_rate,momentum,we,old_grads)
            costs.append(cost)
        j = 0
        for i in reversed (costs):
            print("Epoch: {} ----------> Loss: {} ".format(j+1,i))
            j+=1
        return we,costs[::-1],epochs

    def random_index(sample_size):
        random_index = []
        for i in range(sample_size):
```

```
        index = random.randint(0,46500)
        random_index.append(index)
    return random_index

def pick_sample(data,label,sample_size):
    sample = []
    labels = []
    sample_index = random_index(sample_size)
    for i in sample_index:
        sample.append(data[i])
        labels.append(label[i])
    return sample,labels

def predict(words,output):
    pred_rows = []
    for i in range(len(output)):
        word_index = output[i].argsort()[-10:][::-1]
        pred_words = []
        for word in word_index:
            pred_words.append(str(words[word].decode("utf-8")))
        pred_rows.append((pred_words))
    return pred_rows

def print_preds(random_sample,test_label,pred_rows,words):
    for i in range(len(random_sample)):
        tri = "sample trigram: "
        for j in range(len(random_sample[i])):
            tri+=str(words[random_sample[i][j]].decode("utf-8"))+" "
        tri += " ----> label: " + str(words[test_label[i]].decode("utf-8"))
        print(tri)
        print("Top 10 predictions: ",pred_rows[i])

def plot_losses(losses,epochs,titles):
    xlabel("epoch")
    ylabel("loss")
    title(titles)
    plot(epochs,losses)
############################ Test with different D and P values
######################
```

```
    """epoch changed to 5 in order to save tim TA while running main idea is to show code is
running
    (default is 30 if you want to try you can change by yourself 5th parameter)"""

    we,costs,epochs = train(x_train,y_train,32,256,5,1000,0.15,0.85)
    """we,costs,epochs = train(x_val,y_val,32,256,30,232,0.15,0.85)"""
    plot_losses(costs,epochs,"D=32 - P=256")

    """this part is commented in order to save time for TA to while running

    we,costs,epochs = train(x_train,y_train,16,128,30,1000,0.15,0.85)
    we,costs,epochs = train(x_val,y_val,16,32,30,232,0.15,0.85)
    plot_losses(costs,epochs,"D=16 - P=128")

    we,costs,epochs = train(x_train,y_train,8,64,30,1000,0.15,0.85)
    we,costs,epochs = train(x_val,y_val,16,32,30,232,0.15,0.85)
    plot_losses(costs,epochs,"D=8 - P=64")"""

    ########################## Make Predictions with Test Data
##############################
    random_sample,test_label = pick_sample(x_test,y_test,200)
    random_sample_vector = input_vector(random_sample)
    random_sample_vector = np.squeeze(random_sample_vector,axis=1)
    random_sample = random_sample[:5]
    test_label  = test_label[:5]

    _,_,output = forward(random_sample_vector, we)
    output = output[:5]

    pred_rows = predict(words,output)
    print_preds(random_sample,test_label,pred_rows,words)


import sys

question = sys.argv[1]

def ataberk_cakir_21703127_hw1(question):
    if question == '1' :
        print("Question 1")
        q1()
```

```
    elif question == '2' :
        print("Question 2")
        q2()

ataberk_cakir_21703127_hw1(question)
```