EEE-443 Neural Networks Project-3

Name: Ayberk Yarkın

Surname: Yıldız

Id: 21803386

Date: December 28, 2021

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Question 1

In this question, it is asked to implement an autoencoder neural network with a single hidden layer for the unsupervised feature extraction from natural images. The cost function to be minimized is given as:

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

I wrote the code to save my results in images that will save them in the same folder with the code script.

a) For part a, 200 random sample patches in RGB format are displayed. Additionally, the respective normalized versions of the same patches in grayscale format are displayed after evaluating the required steps. Results are shown below:

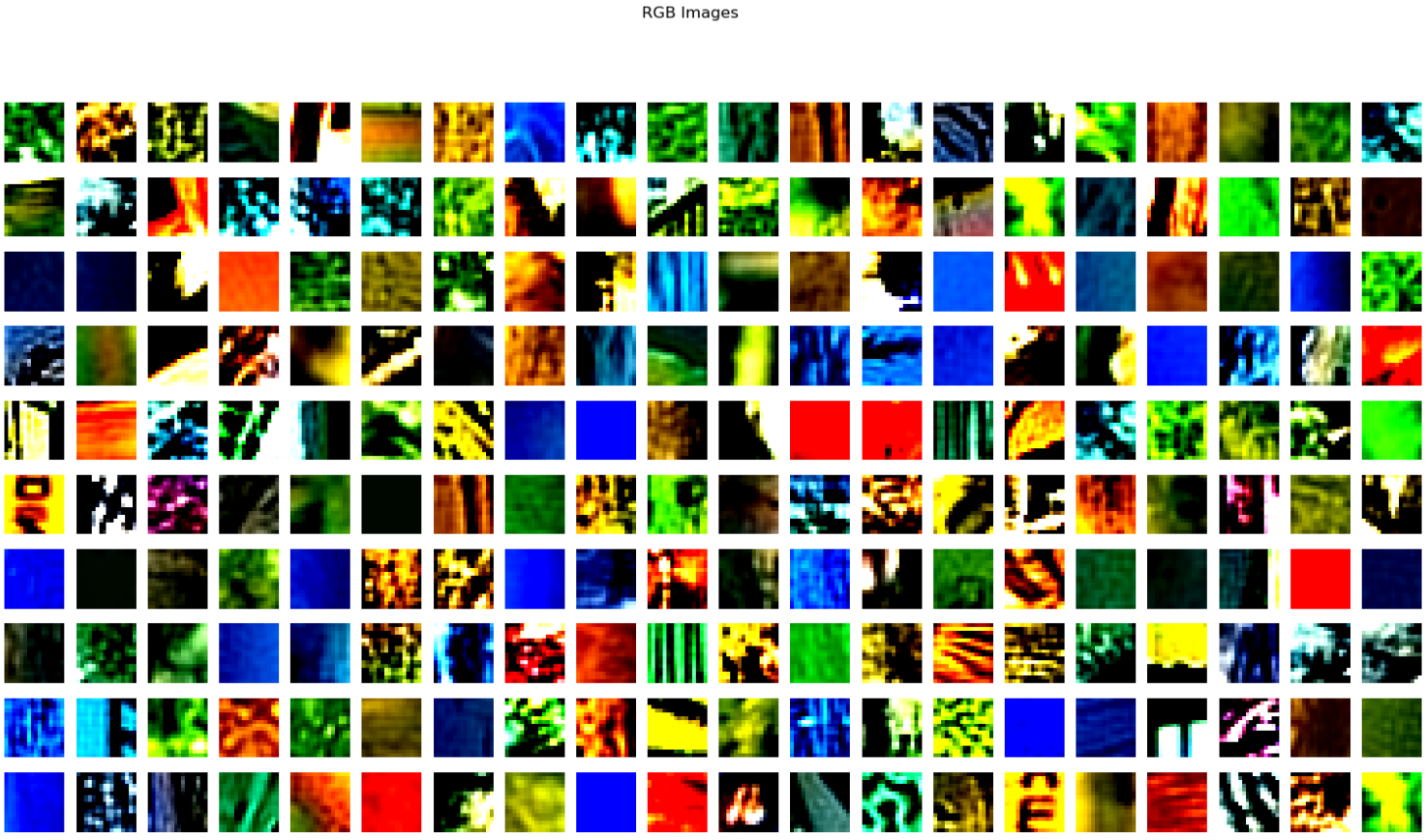

Figure 1: 200 random RGB images of the dataset

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Figure 2: 200 random grayscaled images of the dataset

In the grayscale format, the shapes are the same with the ones in RGB format. Their only difference is that the grayscale formatted images are colorless. Images in grayscale format are easier to see and much more defined and apparent due to their colorless feature.

b) In this part, the required code is written for the network. The code has initialization according to the requirements given in the homework manual as:

$$w_o = \sqrt{\left(\frac{6}{L_{pre} + L_{post}}\right)} \quad (1)$$

where the interval is,

$$weights, bias = [-w_o, w_o] \quad (2)$$

The aeCost(W_e, data, params) function calculates the cost and its partial derivatives J and J_{grad} and returns them. Our weights are determined as:

$$W_e = [W_1\ W_2\ b_1\ b_2] \quad (3)$$

where the weights W_1, W_2 are transposes of each other due to the autoencoder fact. One of them is encoder and the other is the decoder.

Our data is at the size of $L_{in} \times N$ and $params$ include the parameters: $L_{in},\ L_{hid},\ lambda,\ beta,\ rho$. Training is done using the sigmoid function and J and J_{grad} are used in gradient descent solver to minimize the cost.

The average activation term $\hat{\rho}_b$ in the given cost equation is calculated as:

$$\hat{\rho}_b = \frac{1}{N}\sum_{n=0}^{N} h_n \quad (4)$$

The required hidden layer and lambda values with the optimized other parameters of the network and the cost function are:

Parameter	Value
$\boldsymbol{L_{in}}$	256
$\boldsymbol{L_{hid}}$	64
$\boldsymbol{\lambda}$	5×10^{-4}
$\boldsymbol{\beta}$	3
$\boldsymbol{\rho}$	0.03
$\boldsymbol{\alpha}$ **(Momentum Coefficient)**	0.8
$\boldsymbol{\eta}$ **(Learning Rate)**	0.1
Batch	32
Epoch	200

Table 1: Optimized Parameters for Question 1

Next part shows the results.

c) The first layer of connection weights for the neurons in the hidden layer is shown below:

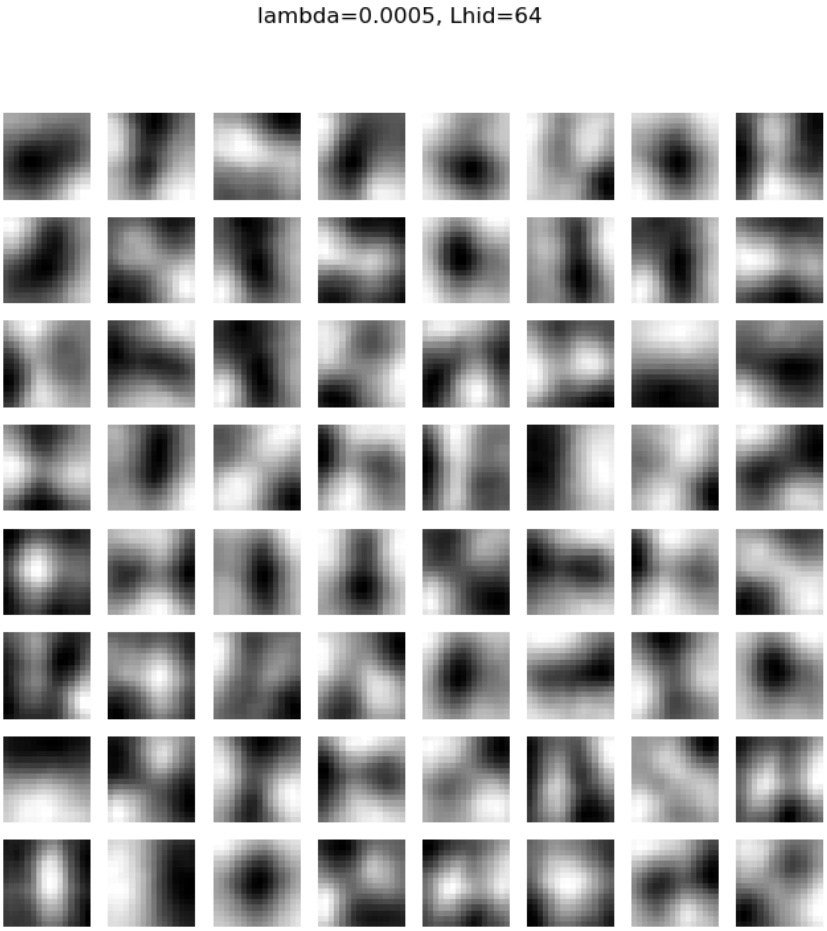

Figure 3: Images of first layer connection weights

Observing the result in Figure 3, there are variations in the images. There are some lines in different orientations, some curved lines and some different shaped parts. These images can combinely create a natural image, but they are not visualizing a natural image specifically. They are not representative of the natural images. The lines and curved lines can be combined and create a square, rectangle or a circle image in the dataset. Or the different shaped parts can be combined and create much complex shaped images in the dataset.

d) In this part, network is retrained with 3 different values of lambda and hidden layer units with determined low, medium and high values. β and ρ values are fixed. The intervals required for this purpose are: $L_{hid} \in [10\ 100]$ and $\lambda \in [0\ 10^{-3}]$. A total 9 combination of these values can be created. I determined my values as: $L_{hid} \in [16, 49, 100]$ and $\lambda \in [0, 10^{-5}, 10^{-3}]$. The hidden layer feature results can be seen below:

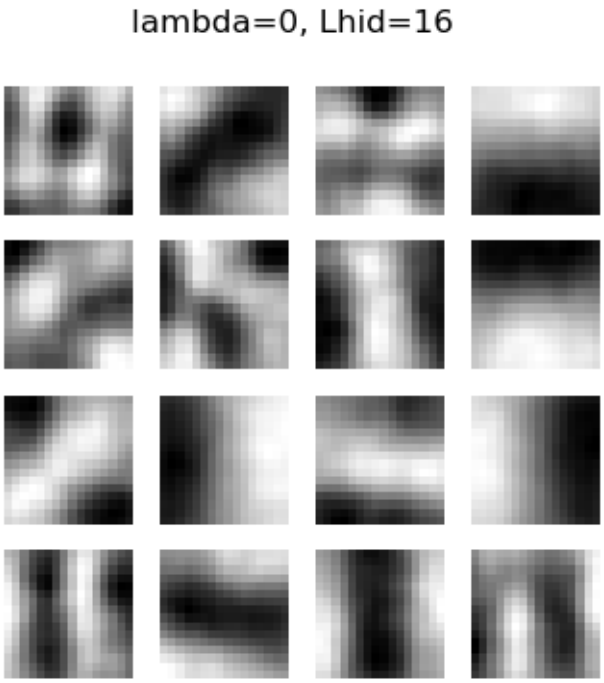

Figure 4: $\lambda = 0, L_{hid} = 16$

Figure 5: $\lambda = 0, L_{hid} = 49$

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Figure 6: $\lambda = 0, L_{hid} = 100$

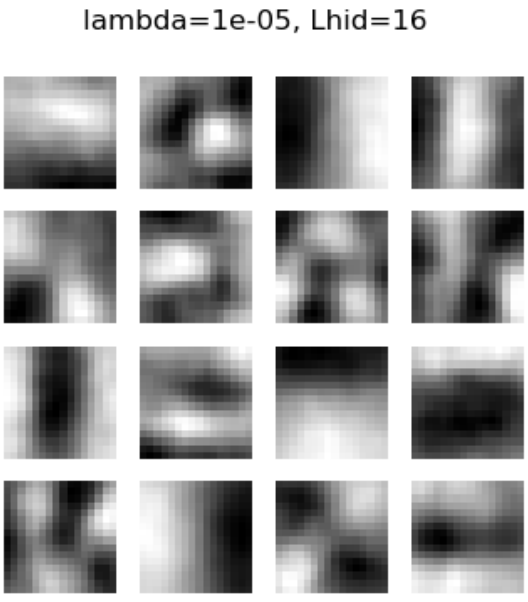

Figure 7: $\lambda = 10^{-5}, L_{hid} = 16$

Figure 8: $\lambda = 10^{-5}, L_{hid} = 49$

Figure 9: $\lambda = 10^{-5}, L_{hid} = 100$

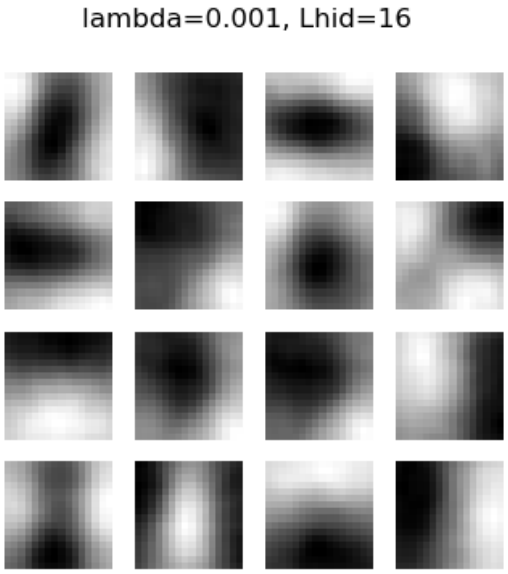

Figure 10: $\lambda = 10^{-3}, L_{hid} = 16$

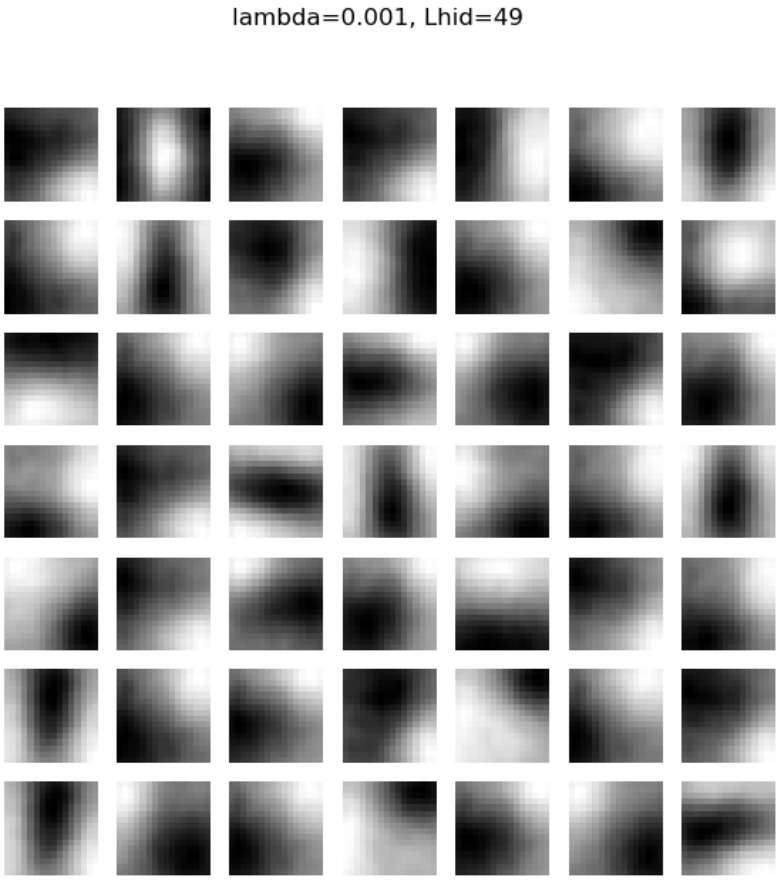

Figure 11: $\lambda = 10^{-3}, L_{hid} = 49$

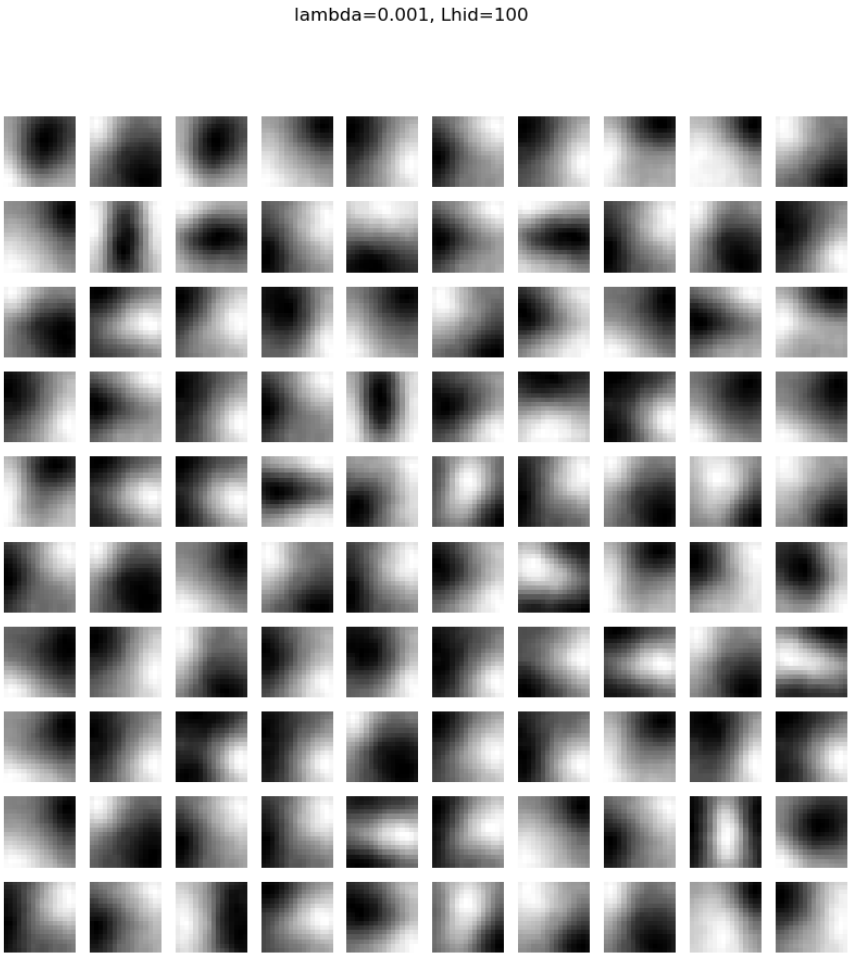

Figure 12: $\lambda = 10^{-3}, L_{hid} = 100$

Observing the results, in general, when L_{hid} is increases, different features are extracted that is a positive feature. Negatively, increasing L_{hid} very much can result in undesired and redundant features and can also result in overfitting.

For λ, increasing it can cause underfitting, and the lower values of λ cause overfitting, such as when $\lambda = 0$ since there is no regularization there is overfitting.

Therefore, λ should be not too small or large, and L_{hid} can be as large as possible but considering the much larger values, larger than 100 which is not in our interval in this case, can cause overfitting. The best combination is when $\lambda = 10^{-3}\ and\ L_{hid} = 100$. $\lambda = 10^{-3}$ is not considered as large in our case, but if it was possible to have a bigger λ, when $\lambda = 10^{-1}$ for example, there would be underfitting.

Question 2

In this question, the CNN models are introduced. One of the CNN models are introduced in Python, the other one is introduced on either TensorFlow or PyTorch frameworks. I chose the PyTorch framework. The jupyter notebook results are shown at the end of the report.

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

a) In the CNN notebook, the convolutional neural network is used on a dataset named CIFAR-10. Firstly, convolutional layer takes place in the network. The two important parameters in this first layer are padding, which reduces the loss of pixels in convolution process, and stride, which determines the windows traversal of the convolution process.

Afterwards, a cat and a dog images are shown to visualize convolution effect. After the backward pass, gradient errors are shown. Max pooling is processed, pool width and height determines the pooling window. Pooling makes the features more generalized and max pooling calculates the max value of the pooling window. Then, after the backward pass, error is displayed.

Then "Fast Layers" are introduced, which are capable of speeding up the operations by using the language C based Cython. Fast convolution and pooling layers are executed. Then, the sandwich layers, that combines the pooling, nonlinearity and convolution into one function, gave some evaluation metrics.

Then, the Three-Layer ConvNet is trained and found its accuracy nearly as 50%, and the first layer filters are displayed.

Lastly, spacial batch normalization and group normalization are mentioned. Spacial batch is a customized and shaped version of batch normalization. Group normalization is an alternative to layer normalization, to have a more effect working with CNN's.

b) As I said at the beginning, I preferred the PyTorch framework for this part. PyTorch uses tensor, which is similar to the arrays, that is used with CPUs and GPUs too, that can increase the efficiency and performence of the network. In the notebook, it creates flatten and twolayerfc functions. Using the PyTorch libraries, a two layer fully connected network is created. One of PyTorch's great advantages is it handles the intermediate values so that the gradient values do not need to be stored.

The sandwich layers in the previous CNN notebook are used to create a three layer network. Then, the two and three layered fully connected networks are trained by two initialization and accuracy calculator and training functions.

API Model of the PyTorch uses its functions for layers and replaces the previous functions in the network, that the model is used to code three layer CNN and a two layer fully connected network. Afterwards, the Sequential API is used to recreate and retrain the previous two layer fully connected network and three layer CNN. This model is very important and very convenient that can replace other training functions much easier. The results of the trained CNN and fully connected network are better than before.

Lastly, the coded model is ready, which has three convolutional models, which are merged into a PyTorch Sequential model. It has batch normalization before convolutional part and there is max pooling at the end. It is asked to get an accuracy higher than 70% by playing

with the given network. I managed to get an accuracy of The explanations and results are shown at the end of the PyTorch notebook at the end of the report.

Question 3

In this question, a human activity is classified from movement signals. The testing data contain time series of T = 150 units. The task is handled by using the first layers as RNN, LSTM and GRU layers for the following parts of the question.

Initialization is done by Xavier Uniform distribution, which is given as:

$$w_o = \sqrt{\left(\frac{6}{L_{pre} + L_{post}}\right)} \quad (5)$$

where the interval is,

$$weights, bias = [-w_o, w_o] \quad (6)$$

The forward pass and backward pass for the multilayer perceptron are shown below:

$$h = \phi(x.weights + bias) \quad (7)$$

$$\frac{dC}{dweights} = h^T.\delta \quad (8)$$

$$\frac{dC}{dbias} = [1]_{1\times L}.\delta \quad (9)$$

Since we are using RNN, LSTM and GRU for this question, δ error gradient is updated at each layer.

For activations, hidden multi-layer perceptrons use ReLU, output layer uses sigmoid function. Cross-entropy error function is used for errors. For backpropogation, BPTT is used for the three different RNN, LSTM and GRU layers. Additionally, momentum is used to get better and more accurate results. The update equation and the update after momentum equation are shown below:

$$\Delta weights = \sum_{t=t_0}^{T} \frac{dC(t)}{dweights} \quad (10)$$

$$weights \ -= \ \eta.\Delta weights + \alpha.weights_{momentum}$$

I wrote the code to save my results in images that will save them in the same folder with the code script.

a) In this part, network is trained by using the first layer as Recurrent layer.

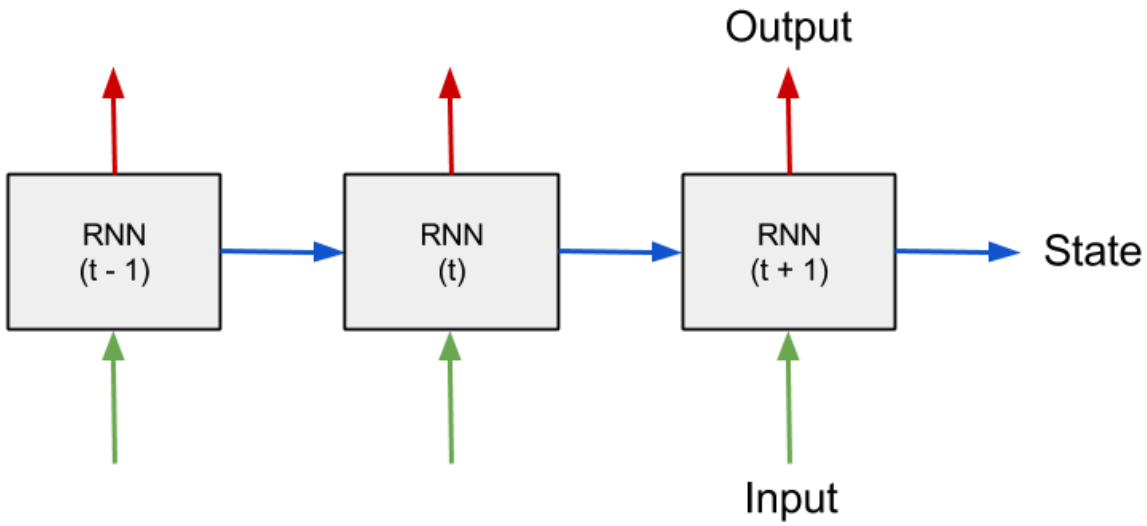

Figure 13: Recurrent layer architecture

RNN's are like the multi-layer perceptron with some changes. It is an extension of multi-layer perceptron that takes the input and also the past output as an input. The respective derivation for our network can be shown as:

$$h(t) = \tanh(x(t) * weights_{ih} + h(t-1) * weights_{hh} + bias) \quad (11)$$

The results are shown below:

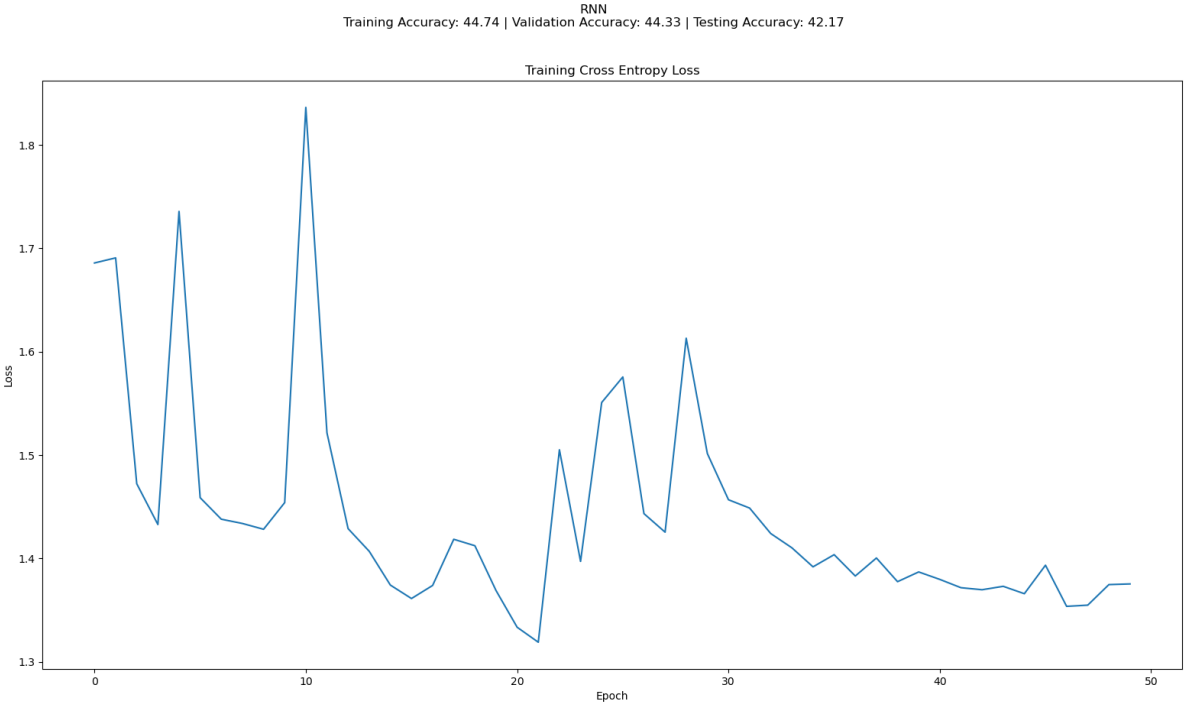

Figure 14: RNN Training Cross Entropy Loss

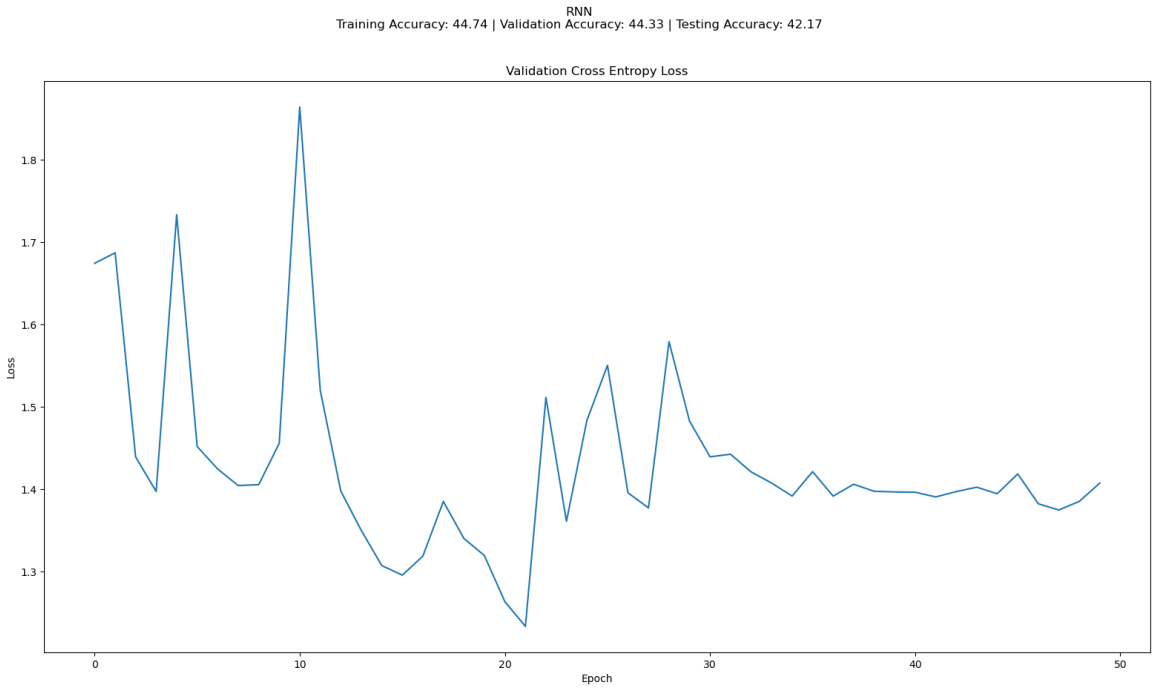

Figure 15: RNN Validation Cross Entropy Loss

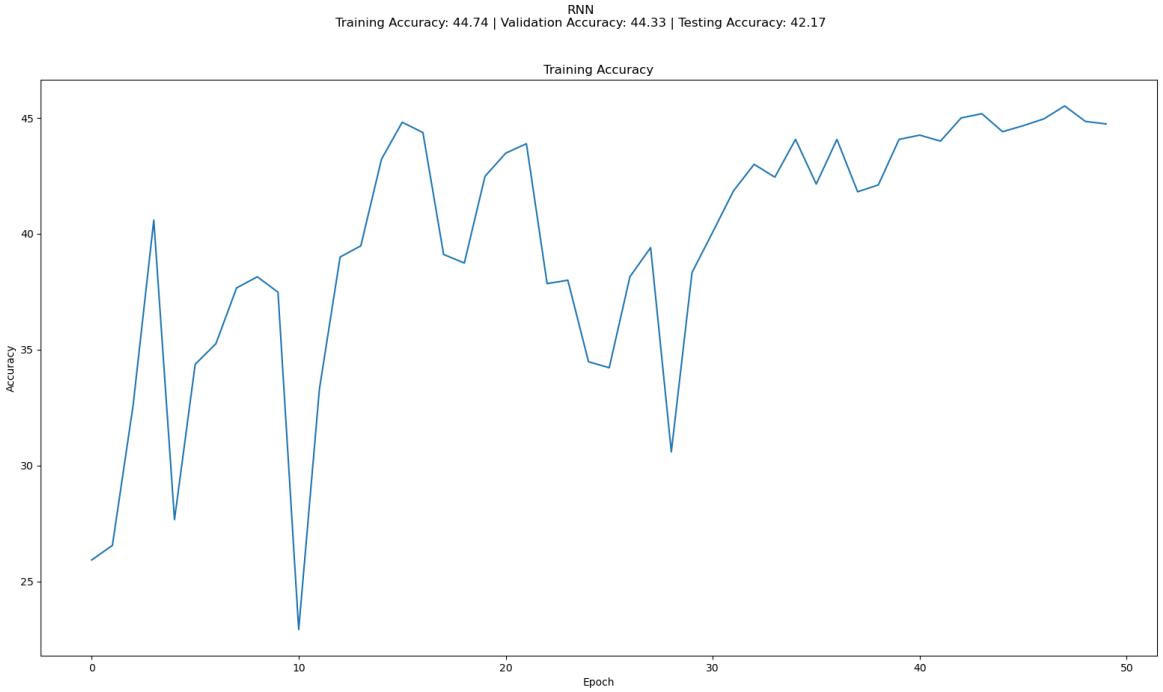

Figure 16: RNN Training Accuracy

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

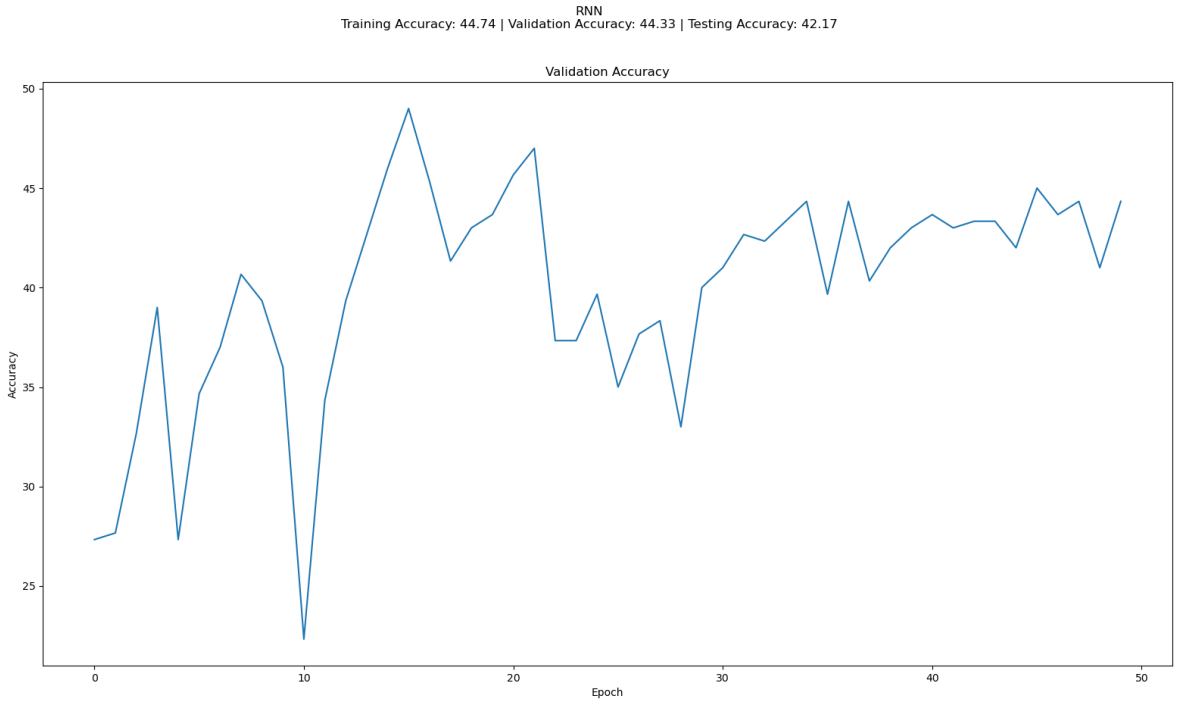

Figure 17: RNN Validation Accuracy

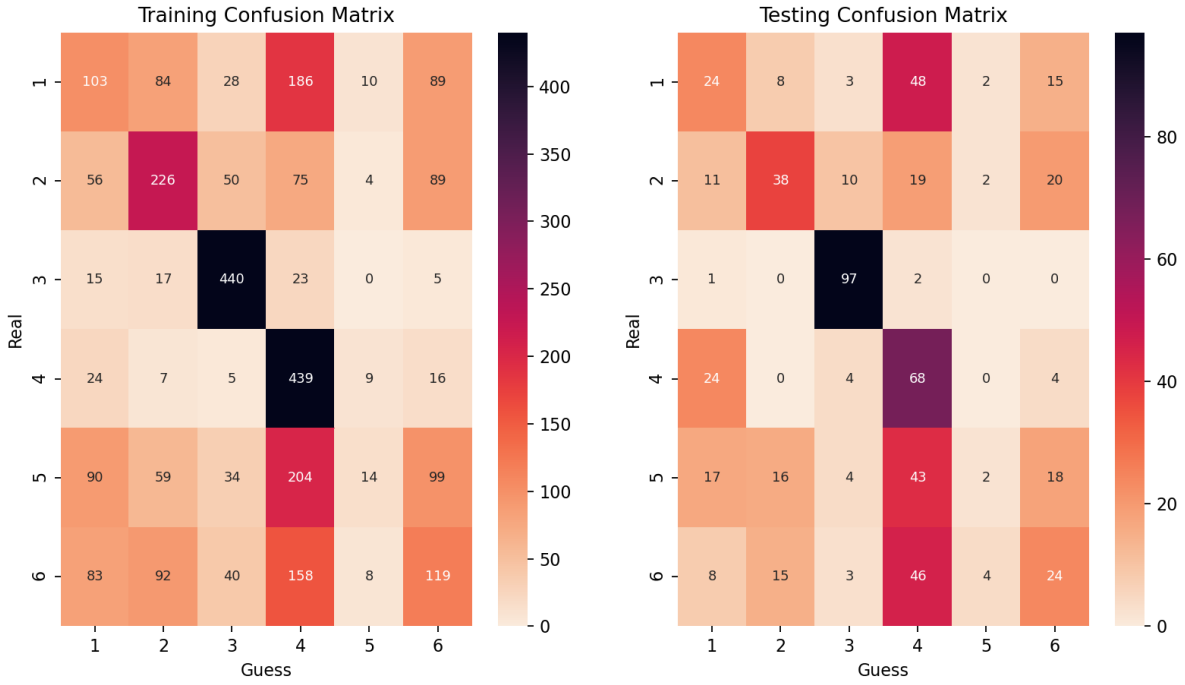

Figure 18: RNN Confusion Matrixes

Training with RNN, there occur some unstability in the loss and accuracy plots. Due to vanishing and exploding gradients problem of RNN, in the long run, it does not always

approaches to a smaller loss when the training goes on, then the network is not learning. Gradients become very high or very low in this case, that results in rapid changes in loss. Having 150 time samples and the cumulative features of the BPTT also results in rapid changes in gradient for RNN.

The parameters can effect the unstability. For example, decreasing the learning rate may decrease the overshooting, but may also result in slow training or even stopping the learning process if it is decreased too much. LSTM and GRU aim to overcome these problems, which will be discussed in the following parts.

b) In this part, network is trained by using the first layer as LSTM.

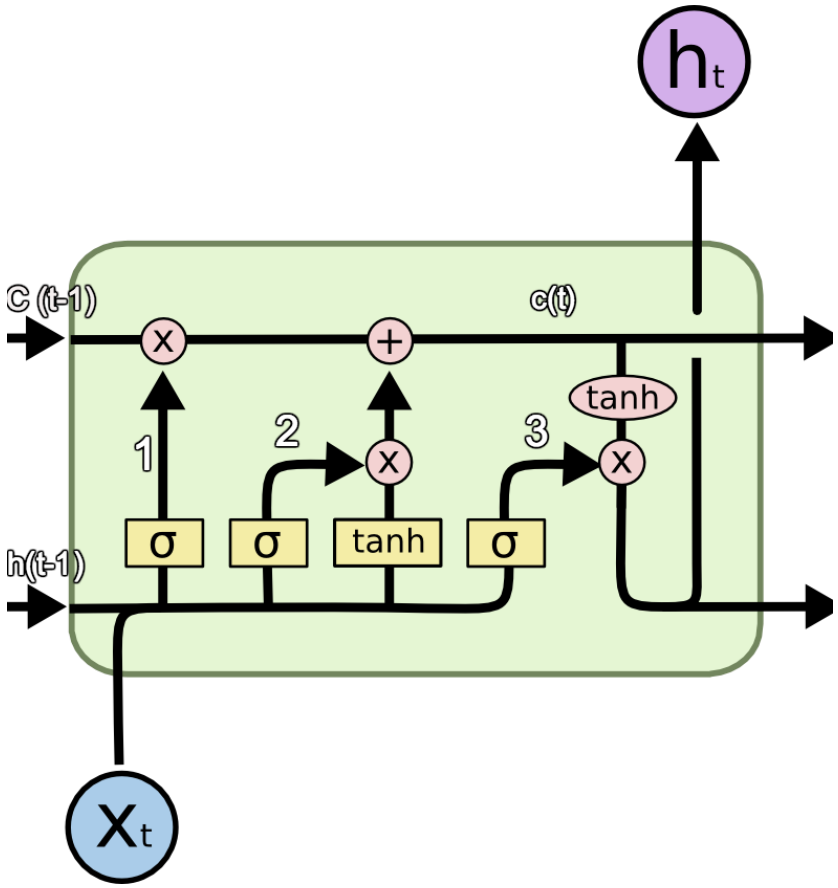

Figure 19: LSTM layer architecture

LSTM solves the problem that occurs in RNN, which is exploding or vanishing gradients. It uses forget gates to forget some of the past memory in the network, which increases the stability. In our network, the equations of LSTM are:

$$hi(t) = \sigma([h(t-1), x(t)].weights_i + bias_i) \quad (12)$$

$$hf(t) = \sigma\big([h(t-1), x(t)].weights_f + bias_f\big) \quad (13)$$

$$ho(t) = \sigma([h(t-1), x(t)].weights_o + bias_o) \quad (14)$$

$$hc(t) = tanh([h(t-1), x(t)].weights_c + bias_c) \quad (15)$$

$$c(t) = hf(t).c(t-1) + hi(t).hc(t) \quad (16)$$

$$h(t) = ho(t).\tanh\big(c(t)\big) \quad (17)$$

The results are shown below:

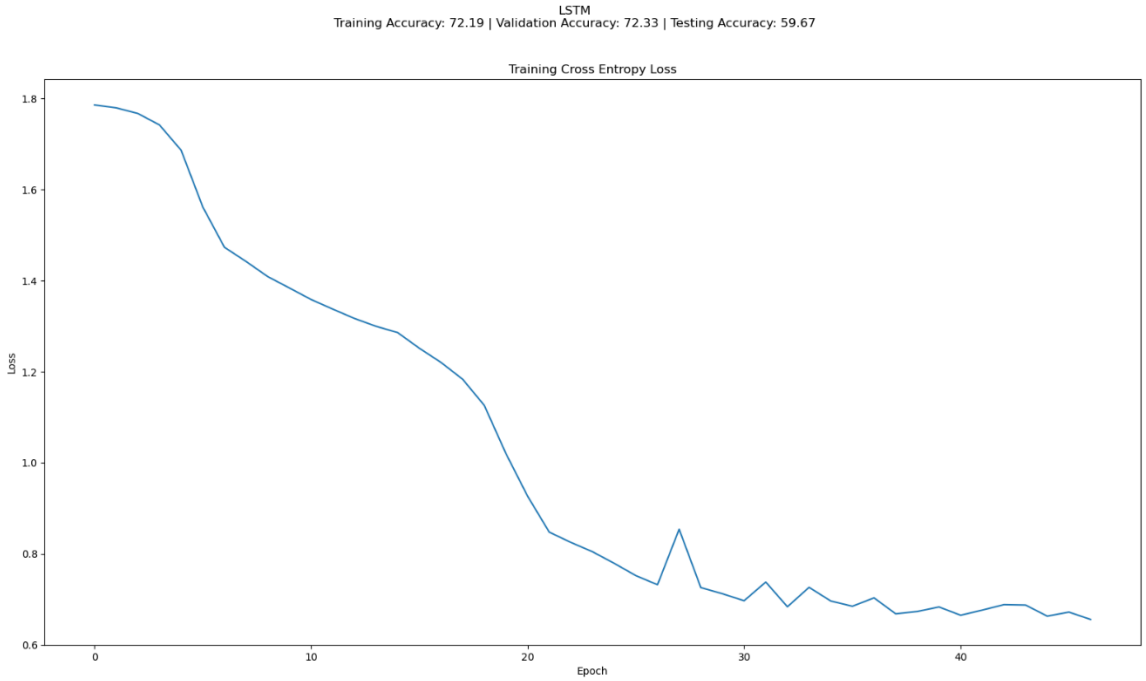

Figure 20: LSTM Training Cross Entropy Loss

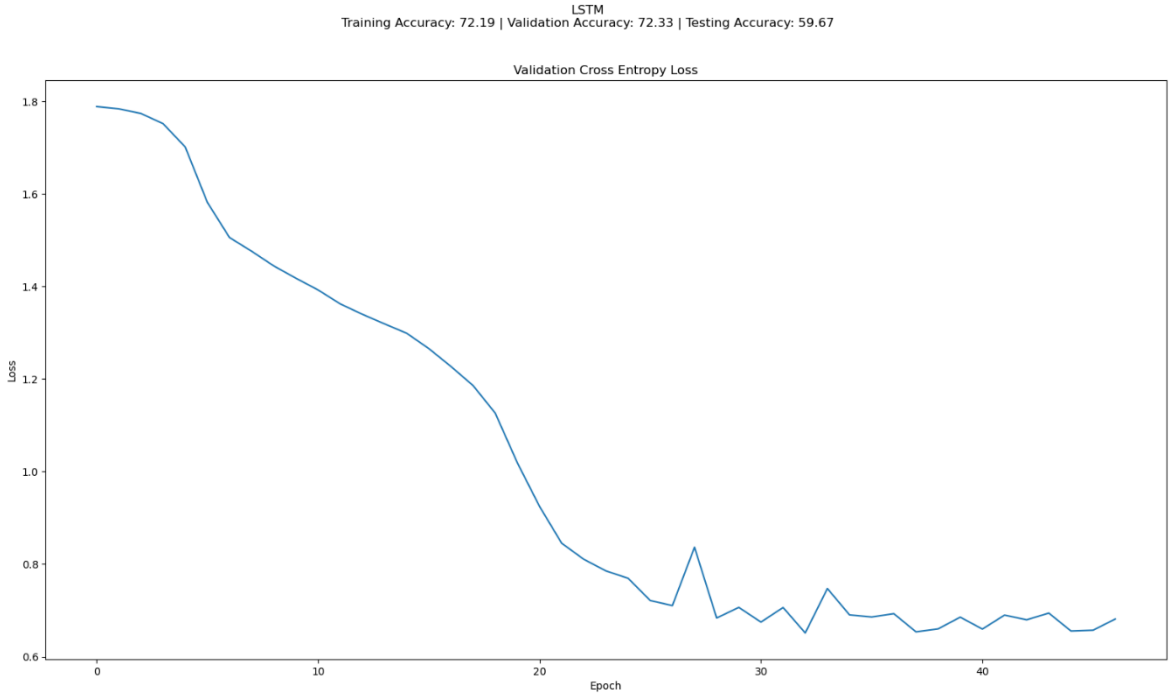

Figure 21: LSTM Validation Cross Entropy Loss

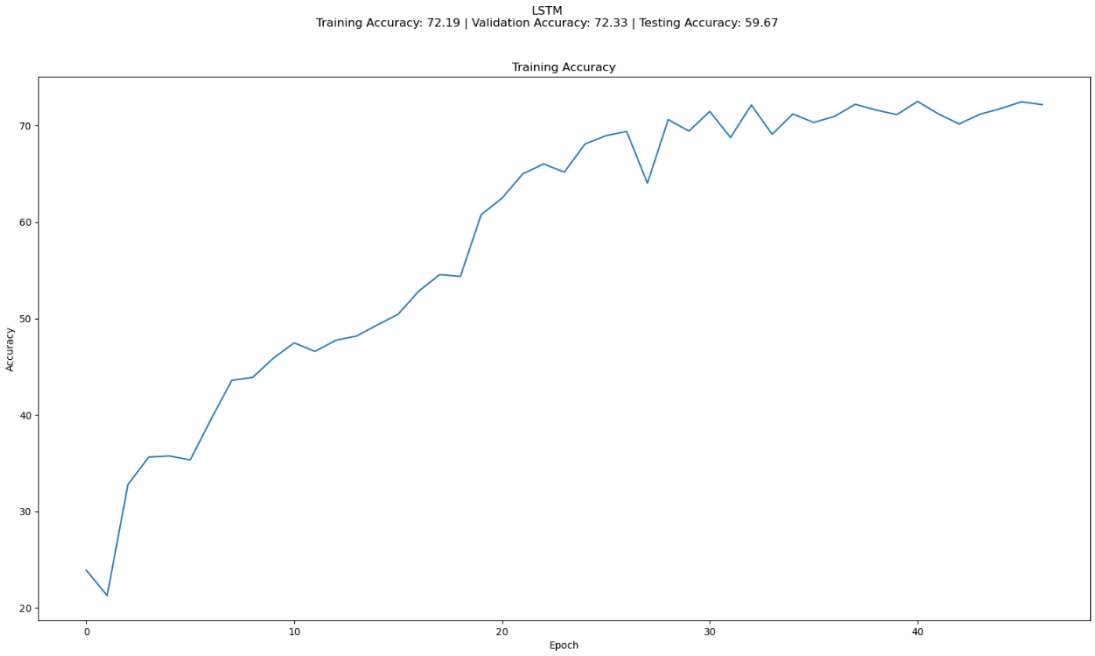

Figure 22: LSTM Training Accuracy

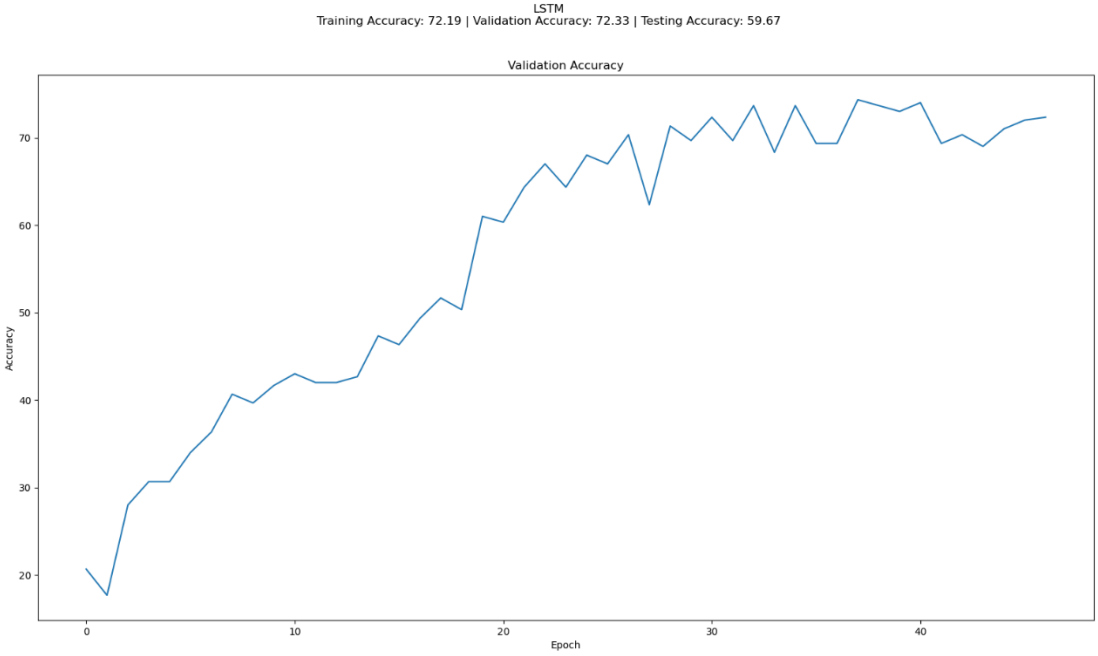

Figure 23: LSTM Validation Accuracy

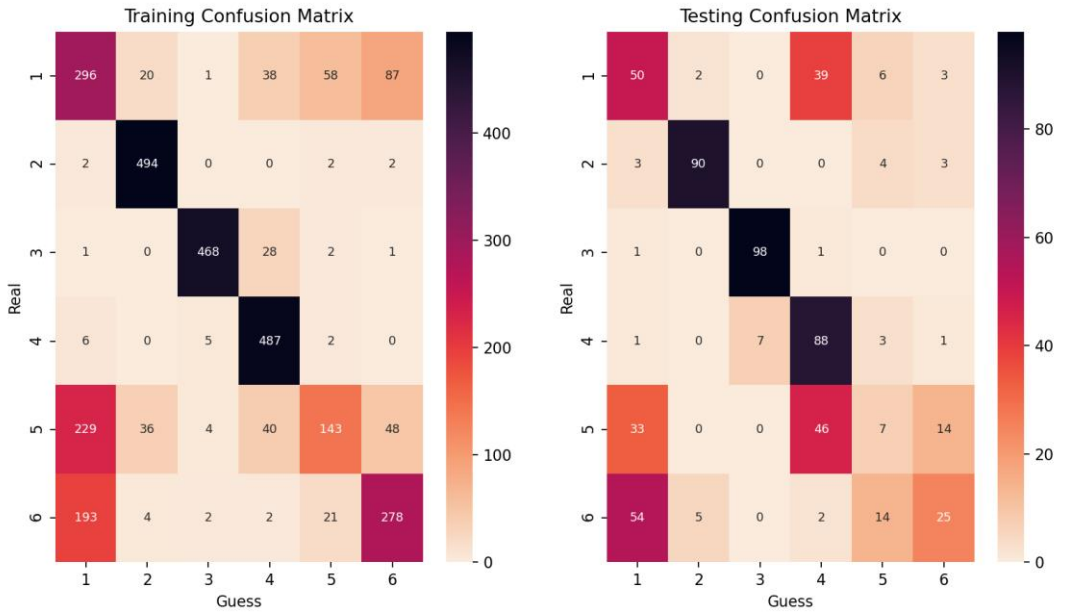

Figure 24: LSTM Confusion Matrixes

Compared to RNN, the network accuracy has increased greatly. Confusion matrices and the test accuracy shows that networks learns and capable of doing correct predictions. Test accuracy becomes 59.67%, where the test accuracy was 42.17% in RNN. Although accuracy has increased, it can be increased more by increasing the learning rate, but it should be fixed in our problem. However, increasing the learning rate too much can also result in exploding gradients.

c) In this last part, network is trained by using the first layer as GRU.

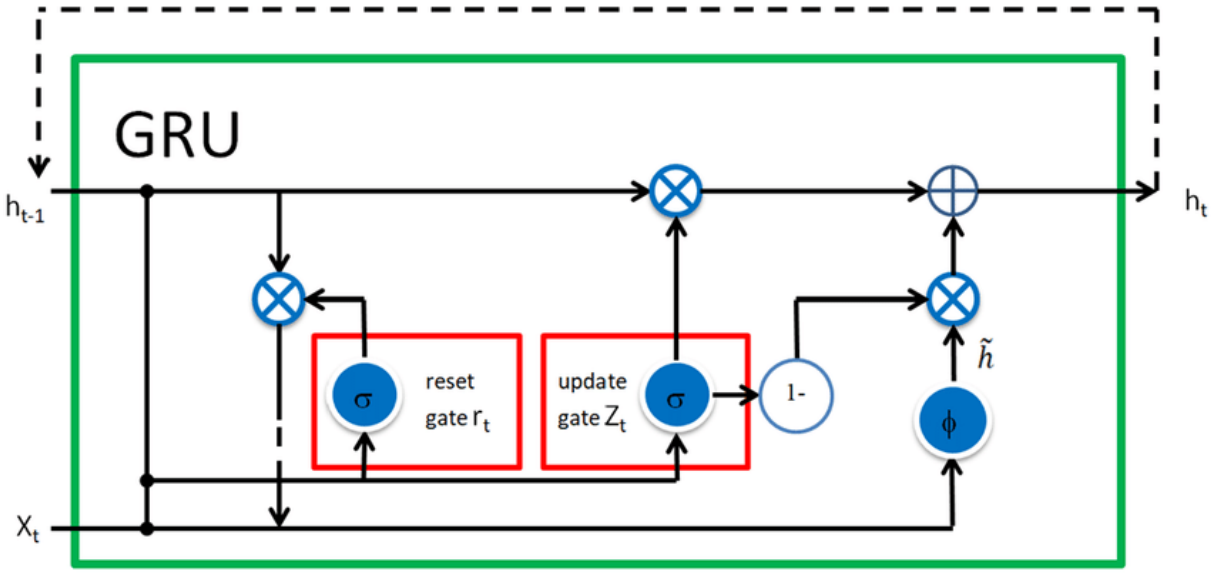

Figure 25: GRU layer architecture

GRU is similar to the LSTM network with slight changes, it has less complexity which is a result of less gates and equations included in it. However, the performance is still great and even better than LSTM that I will talk about after the results. The respective equations for our network are:

$$z(t) = \sigma(x(t).weights_z + h(t-1) * u_z + bias_z) \quad (18)$$

$$r(t) = \sigma(x(t).weigts_r + h(t-1) * u_r + bias_r) \quad (19)$$

$$\tilde{h}(t) = \tanh\big(x(t).weights_h + \big(h(t-1).r(t)\big)u_h + bias_h\big) \quad (20)$$

$$h(t) = \big(1 - z(t)\big).h(t-1) + z(t).\tilde{h}(t) \quad (21)$$

The results are shown below:

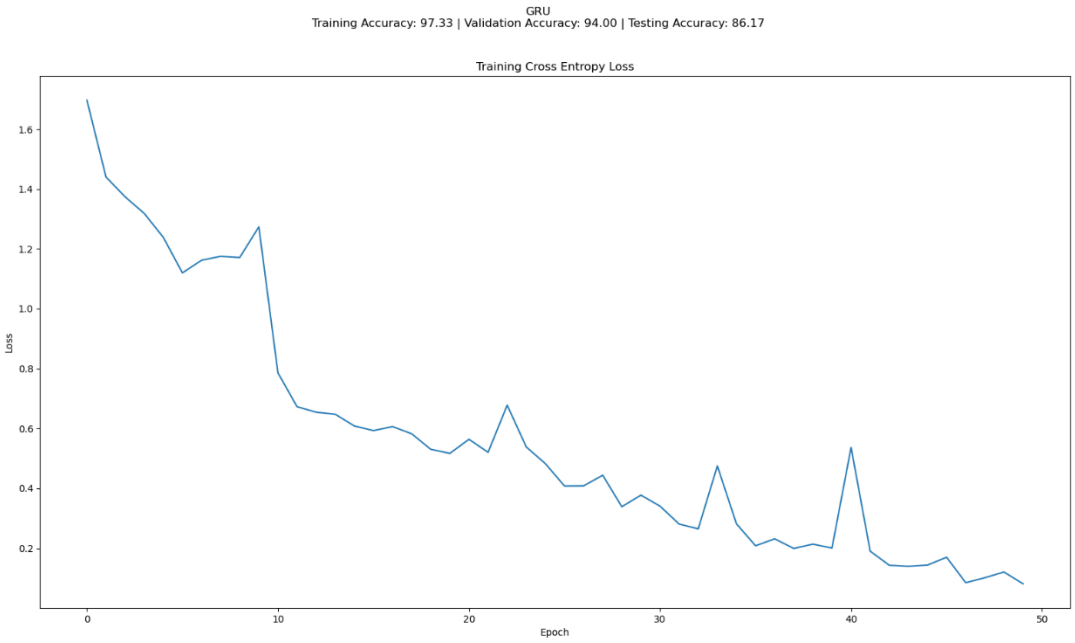

Figure 26: GRU Training Cross Entropy Loss

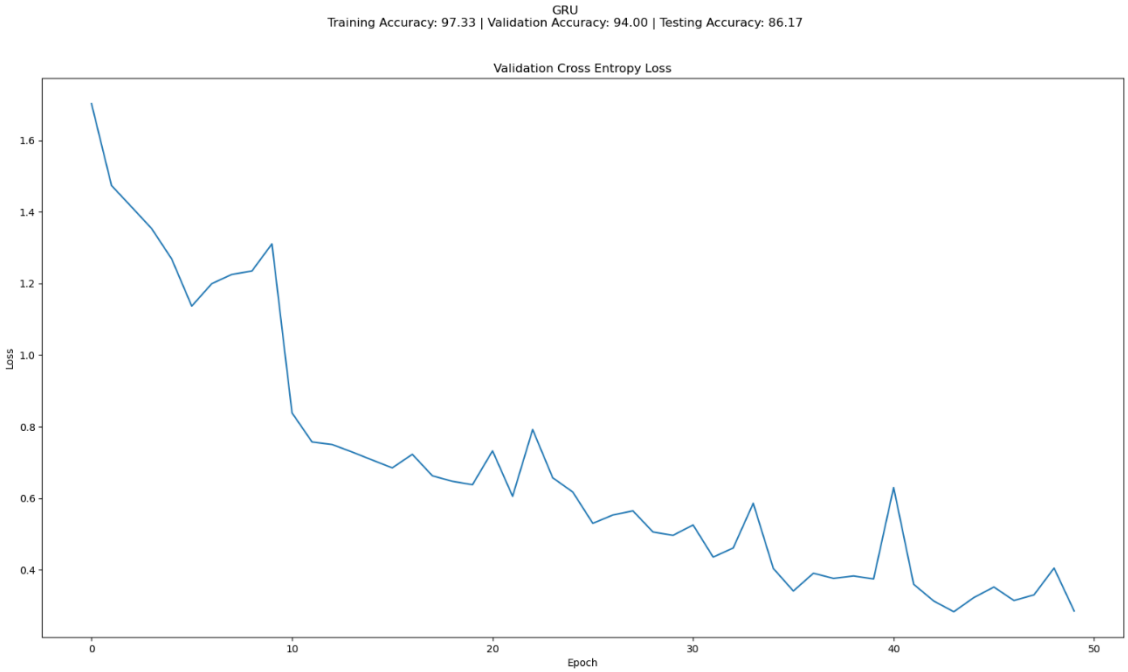

Figure 27: GRU Validation Cross Entropy Loss

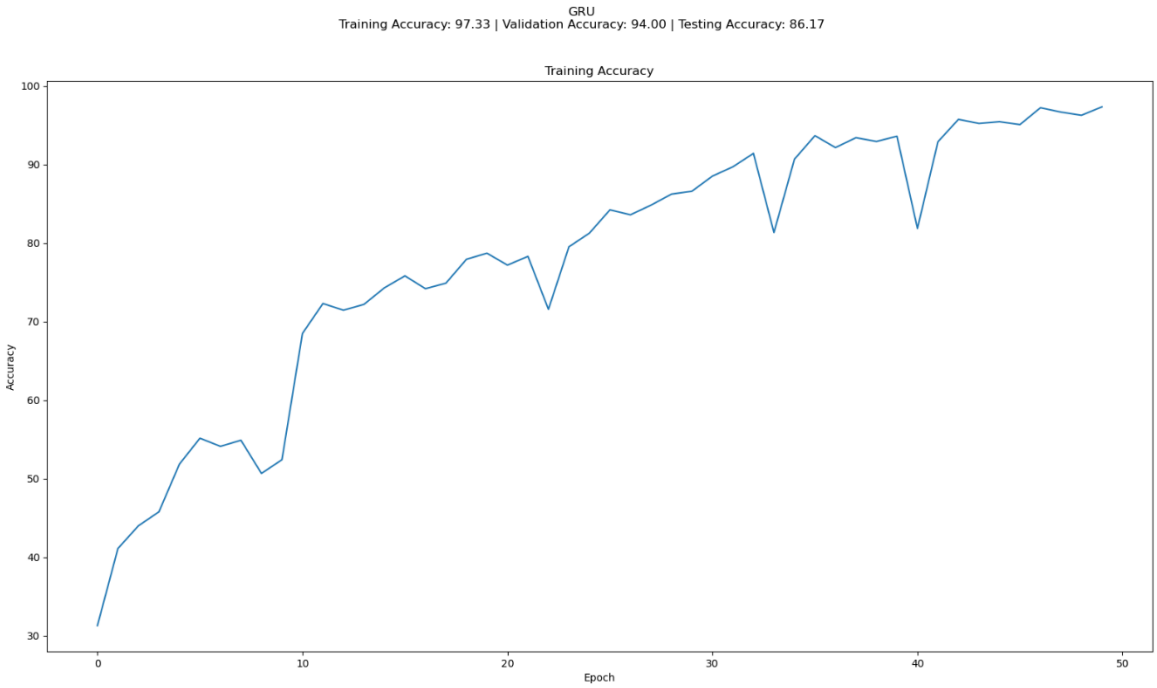

Figure 28: GRU Training Accuracy

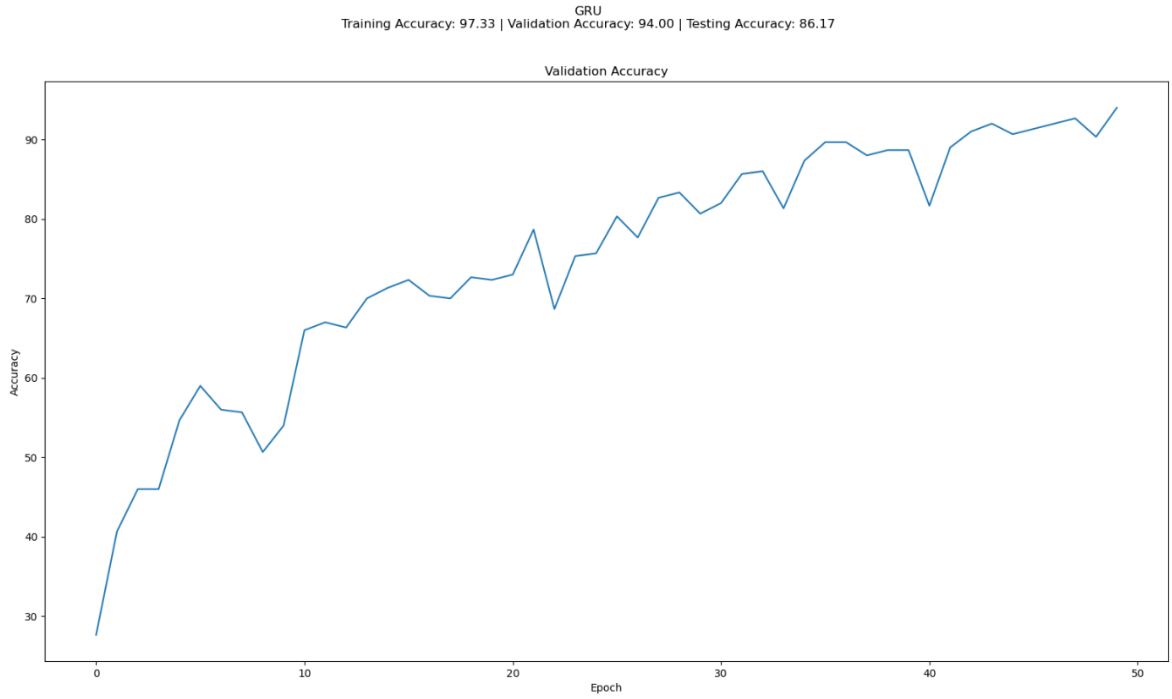

Figure 29: GRU Validation Accuracy

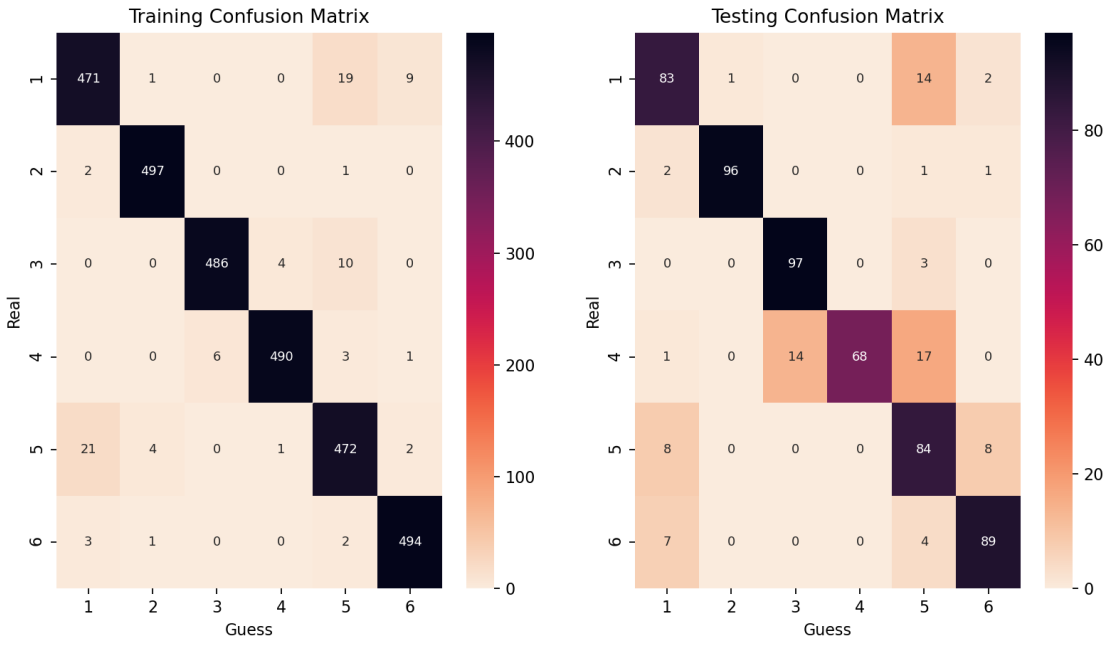

Figure 30: GRU Confusion Matrixes

Ayberk Yarkın Yıldız
21803386
EEE443 Project-3

Compared to the other networks, GRU has the highest accuracy and highest performence. It has a test accuracy of 86.17%. It also had a stable loss at the end, which means that it also solves the problem of vanishing or exploding gradients. Compared to LSTM, GRU's training time is less since it has less gates inside. However, LSTM is more stable and less affected by the gradients' vanishing or exploding. The reason behind this is GRU does not entirely gets rid of the vanishing gradients, just decrease them. LSTM overcomes this situation better, therefore it becomes more stable. Therefore, GRU is built to be efficient, and LSTM is built to have a more stable and accurate network.

Ayberk Yarkın Yıldız
21803386

Appendix

Script That is Run in Command Prompt via "python ayberk_yarkin_yildiz_21803386_hw3.py x", For Both Question 1 and Question 3

```
import sys
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sn
import h5py

def getting(n, bi, tr1, tr2, l, m, b, e, te1, te2, no):
    net = n(bi,no)
    trloss, valoss, tracc, valacc = net.workout3(tr1,tr2,l,m,b,e).values()
    testacc = net.guess3(te1, te2, accur=True)
    return net, trloss, valoss, tracc, valacc, testacc

def guessing(n, tr1, tr2, te1, te2):
    trconf = n.guess3(tr1, tr2, accur=True, conf=True)
    teconf = n.guess3(te1, te2, accur=True, conf=True)
    return trconf, teconf

def lstmproc(w, b, zic, k):
    w = w + zic.T @ k
    b = b + k.sum(axis=0, keepdims=True)
    return w,b

def lstmdu(k, w, big):
    f = k @ w.T[:, :big]
```

```
    return f


def gruproc(a, d, h, w, u, b):
        w = w + a.T @ d
        u = u + h.T @ d
        b = b + d.sum(axis=0, keepdims=True)
        return w,u,b


def ifing1(h, b, n, k):
    if k > 0:
        hb = h[:, k-1, :]
    else:
        hb = np.zeros((n,b))
    return hb


def ifing2(h, k):
    if k > 0:
        hb = h[:, k-1, :]
    else:
        hb = 0
    return hb


def graphn(tr, val, te, dat, namee):
    fig = plt.figure(figsize=(20, 10))
    fig.suptitle(str(namee)+"\nTraining Accuracy: {:.2f} | Validation Accuracy: {:.2f} | Testing
Accuracy: {:.2f}\n ".format(tr[-1], val[-1], te))
    plt.plot(dat)
    plt.xlabel("Epoch")
```

```
def graphm(trconf, testconf):
    plt.figure(figsize=(20, 10), dpi=160)
    plt.subplot(1, 2, 1)
    sn.heatmap(trconf, annot=True, annot_kws={"size": 8}, xticklabels=[1, 2, 3, 4, 5, 6], yticklabels=[1, 2, 3, 4, 5, 6], cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Training Confusion Matrix")
    plt.ylabel("Real")
    plt.xlabel("Guess")
    plt.subplot(1, 2, 2)
    sn.heatmap(testconf, annot=True, annot_kws={"size": 8}, xticklabels=[1, 2, 3, 4, 5, 6], yticklabels=[1, 2, 3, 4, 5, 6], cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Testing Confusion Matrix")
    plt.ylabel("Real")
    plt.xlabel("Guess")


def calcul(a,b,f,k):
    accu = f(a,b,accur=True)
    loso = k(b, f(a,accur=False))
    return accu, loso


def norma(x):
    return (x-x.min())/(x.max()-x.min())


def dis(lrate, mom, epoch, batch, rho, beta, lamda, Lin, Lhid, x, t, d):
    params = {"rho": rho, "beta": beta, "lamda": lamda, "Lin": Lin, "Lhid": Lhid}
    autoencoder = x()
    wson = norma(autoencoder.workout1(t, params, lrate, mom, epoch, batch)[0][0]).T
    wson = wson.reshape((wson.shape[0], d, d))
    w_dimension = int(np.sqrt(wson.shape[0]))
```

```
    return w_dimension, wson


def imageshowc(w,d,lamda,Lhid):
    fig, ax = plt.subplots(d, d, figsize=(d, d))
    fig.suptitle("Question 1, Part c, lambda={}, Lhid={}".format(lamda, Lhid))
    c = 0
    for a in range(d):
        for b in range(d):
            ax[a,b].imshow(w[c], cmap='gray')
            ax[a,b].axis('off')
            c = c + 1
    plt.show()


def imageshowd(w,d,lamda,Lhid):
    fig, ax = plt.subplots(d, d, figsize=(d, d))
    fig.suptitle("Question 1, Part d, lambda={}, Lhid={}".format(lamda, Lhid))
    c = 0
    for a in range(d):
        for b in range(d):
            ax[a,b].imshow(w[c], cmap='gray')
            ax[a,b].axis('off')
            c = c + 1
    plt.show()


def q1():


    class AE():
        def initialize1(self, Lin, Lhid):
            Lout = Lin
```

```
        W1 = np.random.uniform(-(np.sqrt(6)/np.sqrt(Lin + Lhid)),(np.sqrt(6)/np.sqrt(Lin + Lhid)), size=(Lin,Lhid))

        b1 = np.random.uniform(-(np.sqrt(6)/np.sqrt(Lin + Lhid)),(np.sqrt(6)/np.sqrt(Lin + Lhid)), size=(1, Lhid))


        W2 = W1.T

        b2 = np.random.uniform(-np.sqrt(6)/np.sqrt(Lhid + Lout),np.sqrt(6)/np.sqrt(Lhid + Lout), size=(1,Lout))


        We = (W1, W2, b1, b2)

        momWe = (0,0,0,0)


        return We, momWe


    def solver(self, Jgrad, co, We, momWe, lrate, mom):


        W1, W2, b1, b2 = We

        derW1, derW2 = 0,0


        data, h, hder, oder = co

        derloss, dertako2, dertako1, derkel = Jgrad


        derW2 = h.T @ (derloss * oder)+dertako2

        derW1 = data.T @ (hder*((derloss * oder) @ W2.T + derkel)) + dertako1

        derWe = (((derW1.T + derW2)/2).T, (derW1.T + derW2)/2, (hder*((derloss * oder) @ W2.T + derkel)).sum(axis=0, keepdims=True), (derloss * oder).sum(axis=0, keepdims=True))


        We, momWe = self.modify(We, momWe, derWe, lrate, mom)


        return We, momWe
```

```
    def modify(self, We, momWe, derWe, lrate, mom):


        W1, W2, b1, b2 = We

        derW1, derW2, derb1, derb2 = derWe

        momW1, momW2, momb1, momb2 = momWe


        assert(W1 == W2.T).all()

        We = (W1 - (lrate*derW1 + mom*momW1),W2 - (lrate*derW2 + mom*momW2),b1 -
(lrate*derb1 + mom*momb1),b2 - (lrate*derb2 + mom*momb2))

        momWe = (lrate*derW1 + mom*momW1,lrate*derW2 + mom*momW2,lrate*derb1
+ mom*momb1,lrate*derb2 + mom*momb2)


        return We, momWe


    def workout1(self, data, params, lrate, mom, epoch, batch):

        Jl = []


        Lin = params["Lin"]

        Lhid = params["Lhid"]

        We, momWe = self.initialize1(Lin, Lhid)

        for i in range(epoch):


            Jt = 0

            go = 0

            stop = batch


            data = data[np.random.permutation(data.shape[0])]
```

```
        momWe = (0,0,0,0)

        for j in range(int(data.shape[0]/batch)):
            batching = data[go:stop]

            J, Jgrad, co = self.aeCost(We, batching, params)
            We, momWe = self.solver(Jgrad, co, We, momWe, lrate, mom)

            Jt = Jt + J
            go = stop
            stop = stop + batch

        Jt = Jt/(int(data.shape[0]/batch))

        print("Epoch: {}, Loss: {:.3f}".format(i+1, Jt))
        Jl.append(Jt)

    return We, Jl

def aeCost(self, We, data, params):

    nar = data.shape[0]

    W1, W2, b1, b2 = We

    rho = params["rho"]
    beta = params["beta"]
    lamda = params["lamda"]
    Lin = params["Lin"]
```

```
        Lhid = params["Lhid"]


        h, hder = self.sigmoid(data @ W1+b1)


        o, oder = self.sigmoid(h @ W2+b2)


        rhobe = h.mean(axis=0, keepdims=True)


        J = ((0.5*(np.linalg.norm(data-o,axis=1)**2).sum())/nar) +
(0.5*lamda*(np.sum(W1**2) + np.sum(W2**2))) + (beta * (rho*np.log(rho/rhobe) + (1-
rho)*np.log((1-rho)/(1-rhobe))).sum())


        co = (data, h, hder, oder)

        Jgrad = ((o-data)/nar, lamda * W2, lamda * W1, (beta * ((1-rho)/(1-rhobe) -
rho/rhobe)/nar))


        return J, Jgrad, co


    def sigmoid(self, x):

        k = np.exp(x)/(1 + np.exp(x))

        l = k * (1-k)

        return k,l


filename1 = "assign3_data1.h5"

f1 = h5py.File(filename1, 'r')

data = np.array(f1['data'])


""" Part a """

datanew = 0.2126 * data[:, 0] + 0.7152 * data[:, 1] + 0.0722 * data[:, 2]
```

```
    assert datanew.shape[1] == datanew.shape[2]

    dimension = datanew.shape[1]


    datanew = (np.reshape(datanew, (datanew.shape[0], dimension ** 2))) -
(np.reshape(datanew, (datanew.shape[0], dimension ** 2))).mean(axis=1, keepdims = True)

    datanew = 0.1 + 0.8*(norma(np.clip(datanew, - 3 * (np.std(datanew)), 3 *
(np.std(datanew)))))


    trd = datanew


    datanew = np.reshape(datanew, (datanew.shape[0], dimension, dimension))

    data = data.transpose((0,2,3,1))


    """ Part c """


    print("\nQuestion 1 Part c")

    epoch = 200

    mom = 0.8

    rho = 0.03

    beta = 3

    lrate = 0.1

    batch = 32

    lamda = 5e-4

    Lin = trd.shape[1]

    Lhid = 64

    print("\nParameters: rho =",rho,",","beta =",beta,",","lrate =",lrate,",","momentum
=",mom,",","lambda =",lamda,",","batch =",batch,",","Lin =",Lin,",","Lhid =",Lhid,",","epoch
=",epoch,"\n")

    w_dimensioni, wsoni = dis(lrate, mom, epoch, batch, rho, beta, lamda, Lin, Lhid, AE, trd,
dimension)
```

```
""" Part d """


print("\nQuestion 1 Part d")

Lhidl = 16

Lhidn = 49

Lhidh = 100

lamdal = 0

lamdan = 1e-5

lamdah = 1e-3


print("\nlambda =",lamdal,","Lhid =",Lhidl)

w_dimension1, wson1 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidl, AE, trd, dimension)

print("\nlambda =",lamdal,","Lhid =",Lhidn)

w_dimension2, wson2 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidn, AE, trd, dimension)

print("\nlambda =",lamdal,","Lhid =",Lhidh)

w_dimension3, wson3 = dis(lrate, mom, epoch, batch, rho, beta, lamdal, Lin, Lhidh, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidl)

w_dimension4, wson4 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidl, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidn)

w_dimension5, wson5 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidn, AE, trd, dimension)

print("\nlambda =",lamdan,","Lhid =",Lhidh)

w_dimension6, wson6 = dis(lrate, mom, epoch, batch, rho, beta, lamdan, Lin, Lhidh, AE, trd, dimension)

print("\nlambda =",lamdah,","Lhid =",Lhidl)
```

```
    w_dimension7, wson7 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidl, AE,
trd, dimension)
    print("\nlambda =",lamdah,",","Lhid =",Lhidn)
    w_dimension8, wson8 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidn, AE,
trd, dimension)
    print("\nlambda =",lamdah,",","Lhid =",Lhidh)
    w_dimension9, wson9 = dis(lrate, mom, epoch, batch, rho, beta, lamdah, Lin, Lhidh, AE,
trd, dimension)



    """Plots"""
    print('\nTo continue executing, after observation please close the plots')
    print("\nQuestion 1 Part a Plot")
    fig1, ax1 = plt.subplots(10,20,figsize=(20,10))
    fig1.suptitle("Question 1, Part a, RGB Images")
    fig2, ax2 = plt.subplots(10, 20, figsize=(20, 10))
    fig2.suptitle("Question 1, Part a, Grayscale Images")

    for a in range (10):
        for b in range (20):
            c = np.random.randint(0, data.shape[0])
            ax2[a,b].imshow(datanew[c], cmap='gray')
            ax2[a,b].axis("off")

            ax1[a,b].imshow(data[c].astype('float'))
            ax1[a,b].axis('off')

    print('\nTo continue executing, after observation please close the plots')
    plt.show()
```

```
    print("\nQuestion 1 Part c Plot")
    imageshowc(wsoni, w_dimensioni, lamda, Lhid)
    print("\nQuestion 1 Part d Plots")
    imageshowd(wson1, w_dimension1, lamdal, Lhidl)
    imageshowd(wson2, w_dimension2, lamdal, Lhidn)
    imageshowd(wson3, w_dimension3, lamdal, Lhidh)
    imageshowd(wson4, w_dimension4, lamdan, Lhidl)
    imageshowd(wson5, w_dimension5, lamdan, Lhidn)
    imageshowd(wson6, w_dimension6, lamdan, Lhidh)
    imageshowd(wson7, w_dimension7, lamdah, Lhidl)
    imageshowd(wson8, w_dimension8, lamdah, Lhidn)
    imageshowd(wson9, w_dimension9, lamdah, Lhidh)


def q3():
    class Network3():

        def __init__(self,big,num):

            self.num = num
            self.big = big
            self.laybig = len(big)-1
            self.percbig = None
            self.percpar = None
            self.flaypar = None
            self.percmom = None
            self.flaymom = None
            self.initialize3()
```

```
    def initialize3(self):
        num = self.num
        big = self.big
        laybig = self.laybig

        weights = []
        bias = []

        for i in range(1,laybig):
            weights.append(np.random.uniform(-(np.sqrt(6)/np.sqrt(big[1] +
big[i+1])),(np.sqrt(6)/np.sqrt(big[1] + big[i+1])), size=(big[i],big[i+1])))
            bias.append(np.zeros((1, big[i+1])))

        self.percbig = len(weights)
        params = {"weights":weights, "bias":bias}
        mom = {"weights": [0]*self.percbig, "bias": [0]*self.percbig}
        self.percpar = params
        self.percmom = mom

        ne = big[0]
        he = big[1]
        ze = ne + he

        if num == 1:
            weightsih = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size = (ne,he))
            weightshh = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size = (he,he))
            bias = np.zeros((1,he))
```

```
        params = {"weightsih": weightsih, "weightshh": weightshh, "bias": bias}


    if num == 2:

        weightsf = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasf = np.zeros((1,he))

        weightsi = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasi = np.zeros((1,he))

        weightsc = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biasc = np.zeros((1,he))

        weightso = np.random.uniform(-(np.sqrt(6)/np.sqrt(ze+he)),
(np.sqrt(6)/np.sqrt(ze+he)), size = (ze,he))

        biaso = np.zeros((1,he))


        params = {"weightsf": weightsf, "biasf": biasf,"weightsi": weightsi, "biasi":
biasi,"weightsc": weightsc, "biasc": biasc,"weightso": weightso, "biaso": biaso}


    if num == 3:

        weightsz = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))

        uzaz = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size=(he, he))

        biasz = np.zeros((1, he))

        weightsr = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))

        uzar = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)), (np.sqrt(6)/np.sqrt(he+he)),
size=(he, he))

        biasr = np.zeros((1, he))

        weightsh = np.random.uniform(-(np.sqrt(6)/np.sqrt(ne+he)),
(np.sqrt(6)/np.sqrt(ne+he)), size=(ne, he))
```

```
        uzah = np.random.uniform(-(np.sqrt(6)/np.sqrt(he+he)),
(np.sqrt(6)/np.sqrt(he+he)), size=(he, he))

        biash = np.zeros((1, he))


        params = {"weightsz":weightsz, "uzaz": uzaz, "biasz": biasz, "weightsr":weightsr,
"uzar": uzar, "biasr": biasr, "weightsh":weightsh, "uzah": uzah, "biash": biash}


    mom = dict.fromkeys(params.keys(), 0)

    self.flaypar = params

    self.flaymom = mom


  def modify(self, lrate, momc, gradflay, gradperc):


    flaypar = self.flaypar

    flaymom = self.flaymom

    percpar = self.percpar

    percmom = self.percmom


    for k in self.flaypar:

      flaymom[k] = lrate * gradflay[k] + momc * flaymom[k]

      flaypar[k] = flaypar[k] - (lrate * gradflay[k] + momc * flaymom[k])


    for i in range(self.percbig):


        percmom["weights"][i] = lrate*gradperc["weights"][i] + momc *
percmom["weights"][i]

        percmom["bias"][i] = lrate * gradperc["bias"][i] + momc * percmom["bias"][i]


        percpar["weights"][i] = percpar["weights"][i] - (lrate*gradperc["weights"][i] + momc
* percmom["weights"][i])
```

```
            percpar["bias"][i] = percpar["bias"][i] - (lrate * gradperc["bias"][i] + momc *
percmom["bias"][i])


        self.flaypar = flaypar
        self.flaymom = flaymom
        self.percpar = percpar
        self.percmom = percmom

    def ileriperc(self,a,weights,bias,b):
        return self.activ((a @ weights + bias), b)

    def geriperc(self, weights, o, der, chan):

        dweights = o.T @ chan
        dbias = chan.sum(axis=0, keepdims=True)
        chan = der * (chan @ weights.T)
        return dweights, dbias, chan

    def workout3(self, a, b, lrate, momc, batch, epoch):
        traininglossl, validationlossl, trainingaccuracyl, validationaccuracyl = [], [], [], []

        valbig = int(a.shape[0]/10)
        po = np.random.permutation(a.shape[0])
        vala = a[po][:valbig]
        valb = b[po][:valbig]
        a = a[po][valbig:]
        b = b[po][valbig:]
        it = int(a.shape[0]/batch)
```

```
        for i in range(epoch):
            go = 0
            stop = batch
            po = np.random.permutation(a.shape[0])
            a = a[po]
            b = b[po]

            for j in range(it):
                gues, o, der, h, hder, co = self.ilerigo(a[go:stop])

                chan = gues
                chan[b[go:stop] ==1] = chan[b[go:stop] ==1]-1
                chan = chan/batch

                gradflay, gradperc = self.gerigo(a[go:stop], o, der, chan, h, hder, co)

                self.modify(lrate, momc, gradflay, gradperc)

                go = stop
                stop = stop + batch

            trainingaccuracy, trainingloss = calcul(a,b,self.guess3,self.CE)
            validationaccuracy, validationloss = calcul(vala,valb,self.guess3,self.CE)

            print("Epoch: %d | Training Loss: %.3f, Validation Loss: %.3f, Training Accuracy:
%.3f, Validation Accuracy: %.3f"% (i + 1, trainingloss, validationloss, trainingaccuracy,
validationaccuracy))
            traininglossl.append(trainingloss)
            validationlossl.append(validationloss)
```

```
            trainingaccuracyl.append(trainingaccuracy)
            validationaccuracyl.append(validationaccuracy)



            if i>15:
                convergence = sum(validationlossl[-16:-1]) / len(validationlossl[-16:-1])
                if (convergence - 0.001) < validationloss < (convergence + 0.001):
                    print("\nTraining stopped since validation C-E reached convergence.")
                    return {"traininglossl": traininglossl, "validationlossl": validationlossl,
"trainingaccuracyl": trainingaccuracyl, "validationaccuracyl": validationaccuracyl}
        return {"traininglossl": traininglossl, "validationlossl": validationlossl,
"trainingaccuracyl": trainingaccuracyl, "validationaccuracyl": validationaccuracyl}


    def activ(self, a, A):
        if A == "softmax":
            activ = np.exp(a) / np.sum(np.exp(a), axis=1, keepdims=True)
            deriv = None
            return activ, deriv

        if A == "tanh":
            activ = np.tanh(a)
            deriv = 1 - activ**2
            return activ, deriv

        if A == "relu":
            activ = a * (a>0)
            deriv = 1 * (a>0)
            return activ, deriv
```

```
        if A == "sigmoid":
            activ = np.exp(a)/(1 + np.exp(a))
            deriv = activ * (1-activ)
            return activ, deriv

    def ilerirnn(self, a, flaypar):

        ne, te, de = a.shape

        weightsih = flaypar["weightsih"]
        weightshh = flaypar["weightshh"]
        bias = flaypar["bias"]

        hbefore = np.zeros((ne, self.big[1]))
        h, hder = np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            h[:, k, :], hder[:, k, :] = self.activ((a[:, k, :] @ weightsih + hbefore @ weightshh +
bias), "tanh")
            hbefore = h[:, k, :]

        return h, hder

    def gerirnn(self, a, h, hder, chan, flaypar):

        ne, te, de = a.shape
        weightshh = flaypar["weightshh"]
        dweightsih, dweightshh, dbias = 0,0,0
```

```
    for k in reversed(range(te)):
        hbefore = ifing1(h, self.big[1], ne, k)
        hbeforeder = ifing2(hder, k)
        dweightsih = dweightsih + a[:, k, :].T @ chan
        dweightshh = dweightshh + hbefore.T @ chan
        dbias = dbias + chan.sum(axis=0, keepdims=True)
        chan = hbeforeder * (chan@weightshh)

    return {"weightsih": dweightsih, "weightshh": dweightshh, "bias": dbias}

def ilerigo(self,a):

    num = self.num
    percpar = self.percpar
    flaypar = self.flaypar
    o, der = [], []
    h, hder, co = 0,0,0

    if num == 1:
        h, hder = self.ilerirnn(a, flaypar)
        o.append(h[:,-1,:])
        der.append(hder[:,-1,:])

    if num == 2:
        h, co = self.ilerilstm(a, flaypar)
        o.append(h)
        der.append(1)

    if num == 3:
```

```
            h, co = self.ilerigru(a,flaypar)
            o.append(h)
            der.append(1)

        for i in range(self.percbig-1):
            activ, deriv = self.ileriperc(o[-1], percpar["weights"][i], percpar["bias"][i], "relu")
            o.append(activ)
            der.append(deriv)

        gues = self.ileriperc(o[-1], percpar["weights"][-1], percpar["bias"][-1], "softmax")[0]

        return gues, o, der, h, hder, co

    def gerigo(self, a, o, der, chan, h, hder, co):

        num = self.num
        percpar = self.percpar
        flaypar = self.flaypar

        gradflay = dict.fromkeys(percpar.keys())
        gradperc = {"weights": [0] * self.percbig, "bias": [0]*self.percbig}

        for i in reversed(range(self.percbig)):
            gradperc["weights"][i], gradperc["bias"][i], chan =
self.geriperc(percpar["weights"][i], o[i], der[i], chan)

        if num == 1:
            gradflay = self.gerirnn(a, h, hder, chan, flaypar)
        if num == 2:
```

```
            gradflay = self.gerilstm(co, flaypar, chan)
        if num == 3:
            gradflay = self.gerigru(a, co, flaypar, chan)

        return gradflay, gradperc

    def ilerilstm(self, a, flaypar):

        ne, te, de = a.shape

        weightsi, biasi = flaypar["weightsi"], flaypar["biasi"]
        weightsf, biasf = flaypar["weightsf"], flaypar["biasf"]
        weightso, biaso = flaypar["weightso"], flaypar["biaso"]
        weightsc, biasc = flaypar["weightsc"], flaypar["biasc"]

        hbefore, cbefore = np.zeros((ne, self.big[1])), np.zeros((ne, self.big[1]))
        zi = np.empty((ne, te, de + self.big[1]))
        hfi = 0

        hii, hci, hoi, tanhci, ci, tanhcdi, hfdi, hidi, hcdi, hodi = np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])),
np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            zi[:, k, :] = np.column_stack((hbefore, a[:, k, :]))

            hfi, hfdi[:, k, :] = self.activ(zi[:, k, :] @ weightsf + biasf, "sigmoid")
            hii[:, k, :], hidi[:, k, :] = self.activ(zi[:, k, :] @ weightsi + biasi, "sigmoid")
            hci[:, k, :], hcdi[:, k, :] = self.activ(zi[:, k, :] @ weightsc + biasc, "tanh")
```

```
            hoi[:, k, :], hodi[:, k, :] = self.activ(zi[:, k, :] @ weightso + biaso, "sigmoid")


            ci[:, k, :] = hfi * cbefore + hii[:, k, :] * hci[:, k, :]
            tanhci[:, k, :], tanhcdi[:, k, :] = self.activ(ci[:, k, :], "tanh")
            hbefore = hoi[:, k, :] * tanhci[:, k, :]
            cbefore = ci[:, k, :]


            co = {"zi": zi, "ci": ci, "tanhci": (tanhci, tanhcdi), "hfdi": hfdi, "hii": (hii, hidi), "hci":
(hci, hcdi), "hoi": (hoi, hodi)}


        return hbefore, co


    def gerilstm(self, co, flaypar, chan):


        weightsf = flaypar["weightsf"]
        weightsi = flaypar["weightsi"]
        weightsc = flaypar["weightsc"]
        weightso = flaypar["weightso"]


        zi = co["zi"]
        ci = co["ci"]
        tanhci, tanhcdi = co["tanhci"]
        hfdi = co["hfdi"]
        hii, hidi = co["hii"]
        hci, hcdi = co["hci"]
        hoi, hodi = co["hoi"]
        te = zi.shape[1]
```

```
        dweightsf, dweightsi, dweightsc, dweightso, dbiasf, dbiasi, dbiasc, dbiaso =
0,0,0,0,0,0,0,0


        for k in reversed(range(te)):
            cbefore = ifing2(ci, k)


            dci = chan * hoi[:, k, :] * tanhcdi[:, k, :]
            dhfi = dci * cbefore * hfdi[:, k, :]
            dhii = dci * hci[:, k, :] * hidi[:, k, :]
            dhci = dci * hii[:, k, :] * hcdi[:, k, :]
            dhoi = chan * tanhci[:, k, :] * hodi[:, k, :]


            dweightsf, dbiasf = lstmproc(dweightsf, dbiasf, zi[:, k, :], dhfi)
            dweightsi, dbiasi = lstmproc(dweightsi, dbiasi, zi[:, k, :], dhii)
            dweightsc, dbiasc = lstmproc(dweightsc, dbiasc, zi[:, k, :], dhci)
            dweightso, dbiaso = lstmproc(dweightso, dbiaso, zi[:, k, :], dhoi)


            df = lstmdu(dhfi, weightsf, self.big[1])
            di = lstmdu(dhii, weightsi, self.big[1])
            dc = lstmdu(dhci, weightsc, self.big[1])
            do = lstmdu(dhoi, weightso, self.big[1])



            chan = (df + di + dc + do)


        return {"weightsf": dweightsf, "biasf": dbiasf, "weightsi": dweightsi, "biasi": dbiasi,
"weightsc": dweightsc, "biasc": dbiasc, "weightso": dweightso, "biaso": dbiaso}


    def ilerigru(self, a, flaypar):
```

```
        weightsz = flaypar["weightsz"]
        weightsr = flaypar["weightsr"]
        weightsh = flaypar["weightsh"]

        uzaz = flaypar["uzaz"]
        uzar = flaypar["uzar"]
        uzah = flaypar["uzah"]

        biasz = flaypar["biasz"]
        biasr = flaypar["biasr"]
        biash = flaypar["biash"]

        ne, te, de = a.shape
        hbefore = np.zeros((ne, self.big[1]))
        zi, zdi, ri, rdi, htider, htiderd, hi = np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1])), np.empty((ne, te, self.big[1]))

        for k in range(te):
            zi[:, k, :], zdi[:, k, :] = self.activ(a[:, k, :] @ weightsz + hbefore @ uzaz + biasz, "sigmoid")
            ri[:, k, :], rdi[:, k, :] = self.activ(a[:, k, :] @ weightsr + hbefore @ uzar + biasr, "sigmoid")
            htider[:, k, :], htiderd[:, k, :] = self.activ(a[:, k, :] @ weightsh + (ri[:, k, :] * hbefore) @ uzah + biash, "tanh")
            hi[:, k, :] = (1 - zi[:, k, :]) * hbefore + zi[:, k, :] * htider[:, k, :]

            hbefore = hi[:, k, :]

        co = {"zi": (zi, zdi), "ri": (ri, rdi), "htider": (htider, htiderd), "hi": hi}
```

```
        return hbefore, co


    def gerigru(self, a, co, flaypar, chan):

        uzaz = flaypar["uzaz"]
        uzar = flaypar["uzar"]
        uzah = flaypar["uzah"]

        zi, zdi = co["zi"]
        ri, rdi = co["ri"]
        htider, htiderd = co["htider"]
        hi = co["hi"]

        ne, te, de = a.shape
        dweightsz, dweightsr, dweightsh, duzaz, duzar, duzah, dbiasz, dbiasr, dbiash =
0,0,0,0,0,0,0,0,0

        for k in reversed(range(te)):
            hbefore = ifing1(hi, self.big[1], ne, k)

            dzi = chan * (htider[:, k, :] - hbefore) * zdi[:, k, :]
            dhtider = chan * zi[:, k, :] * htiderd[:, k, :]
            dri = (dhtider @ uzah.T) * hbefore * rdi[:, k, :]

            dweightsz, duzaz, dbiasz = gruproc(a[:, k, :], dzi, hbefore, dweightsz, duzaz, dbiasz)
            dweightsr, duzar, dbiasr = gruproc(a[:, k, :], dri, hbefore, dweightsr, duzar, dbiasr)
            dweightsh, duzah, dbiash = gruproc(a[:, k, :], dhtider, hbefore, dweightsh, duzah,
dbiash)
```

```
            chan = (chan * (1 - zi[:, k, :])) + (dzi @ uzaz.T) + ((dhtider @ uzah.T) * (ri[:, k, :] +
hbefore * (rdi[:, k, :] @ uzar.T)))


        return {"weightsz": dweightsz, "uzaz": duzaz, "biasz": dbiasz, "weightsr": dweightsr,
"uzar": duzar, "biasr": dbiasr, "weightsh": dweightsh, "uzah": duzah, "biash": dbiash}


    def guess3(self, a, b=None, accur=True, conf=False):


        guessino = self.ilerigo(a)[0]


        if not accur:
            return guessino


        guessino = guessino.argmax(axis=1)
        b = b.argmax(axis=1)


        if not conf:
            return (guessino == b).mean() * 100


        cla = np.zeros((len(np.unique(b)),len(np.unique(b))))


        for k in range(len(b)):
            cla[b[k]][guessino[k]] = cla[b[k]][guessino[k]] + 1


        return cla


    def CE(self, d, y):
        return np.sum(np.log(y) * -d) / d.shape[0]
```

```
filename3 = "assign3_data3.h5"
f3 = h5py.File(filename3, 'r')

trainx = np.array(f3['trX'])
trainy = np.array(f3['trY'])
testx = np.array(f3['tstX'])
testy = np.array(f3['tstY'])



"""Part a"""

print("Question 3 Part a")
print("Recurrent Layer\n")
epoch3rnn = 50
lrate3rnn = 0.01
batch3rnn = 32
momc3rnn = 0.85
bigrnn = [trainx.shape[2], 128, 32, 16, 6]

net3rnn, traininglosslrnn, validationlosslrnn, trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn = getting(Network3, bigrnn, trainx, trainy, lrate3rnn, momc3rnn, batch3rnn, epoch3rnn, testx, testy, 1)
print("\nTest Accuracy: ", testaccuracyrnn, "\n\n")

trainingconfrnn, testingconfrnn = guessing(net3rnn, trainx, trainy, testx, testy)

"""Part b"""
print("\nQuestion 3 Part b")
```

```
print("LSTM Layer\n")


epoch3lstm = 50

lrate3lstm = 0.01

batch3lstm = 32

momc3lstm = 0.85

biglstm = [trainx.shape[2], 128, 32, 16, 6]


net3lstm, traininglossllstm, validationlossllstm, trainingaccuracyllstm,
validationaccuracyllstm, testaccuracylstm = getting(Network3, biglstm, trainx, trainy,
lrate3lstm, momc3lstm, batch3lstm, epoch3lstm, testx, testy, 2)

print("\nTest Accuracy: ", testaccuracylstm, "\n\n")


trainingconflstm, testingconflstm = guessing(net3lstm, trainx, trainy, testx, testy)


"""Part c"""

print("\nQuestion 3 Part c")

print("GRU Layer\n")


epoch3gru = 50

lrate3gru = 0.01

batch3gru = 32

momc3gru = 0.85

biggru = [trainx.shape[2], 128, 32, 16, 6]


net3gru, traininglosslgru, validationlosslgru, trainingaccuracylgru, validationaccuracylgru,
testaccuracygru= getting(Network3, biggru, trainx, trainy, lrate3gru, momc3gru, batch3gru,
epoch3gru, testx, testy, 3)

print("\nTest Accuracy: ", testaccuracygru, "\n\n")
```

```
    trainingconfgru, testingconfgru = guessing(net3gru, trainx, trainy, testx, testy)


    """ Plots"""
    print('To continue executing, after observation please close the plots')
    print("\nQuestion 3, Part a Plots")
    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn, traininglosslrnn,
"RNN")
    plt.title("Training Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn, validationlosslrnn,
"RNN")
    plt.title("Validation Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn,
trainingaccuracylrnn, "RNN")
    plt.title("Training Accuracy")
    plt.ylabel("Accuracy")
    plt.show()


    graphn(trainingaccuracylrnn, validationaccuracylrnn, testaccuracyrnn,
validationaccuracylrnn, "RNN")
    plt.title("Validation Accuracy")
    plt.ylabel("Accuracy")
    plt.show()


    graphm(trainingconfrnn, testingconfrnn)
```

```
    plt.show()



    print("\nQuestion 3, Part b Plots")
    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, trainingloss llstm, "LSTM")
    plt.title("Training Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, validationloss llstm, "LSTM")
    plt.title("Validation Cross Entropy Loss")
    plt.ylabel("Loss")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, trainingaccuracyllstm, "LSTM")
    plt.title("Training Accuracy")
    plt.ylabel("Accuracy")
    plt.show()

    graphn(trainingaccuracyllstm, validationaccuracyllstm, testaccuracylstm, validationaccuracyllstm, "LSTM")
    plt.title("Validation Accuracy")
    plt.ylabel("Accuracy")
    plt.show()

    graphm(trainingconflstm, testingconflstm)
    plt.show()
```

```
    print("\nQuestion 3, Part c Plots")

    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, trainingloss lgru, "GRU")

    plt.title("Training Cross Entropy Loss")

    plt.ylabel("Loss")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, validationloss lgru, "GRU")

    plt.title("Validation Cross Entropy Loss")

    plt.ylabel("Loss")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, trainingaccuracylgru, "GRU")

    plt.title("Training Accuracy")

    plt.ylabel("Accuracy")

    plt.show()


    graphn(trainingaccuracylgru, validationaccuracylgru, testaccuracygru, validationaccuracylgru, "GRU")

    plt.title("Validation Accuracy")

    plt.ylabel("Accuracy")

    plt.show()


    graphm(trainingconfgru, testingconfgru)

    plt.show()
```

```
question = sys.argv[1]


def ayberk_yarkin_yildiz_21803386_hw3(question):

   if question == '1' :
      q1()

   elif question == '3' :
      q3()

ayberk_yarkin_yildiz_21803386_hw3(question)
```

Question 2