MINI PROJECT

1 Question 1

In this question we are expected to implement an auto-encoder neural network architecture such that the architecture will have a single hidden layer to extract features from natural images in an unsupervised way. In this process the cost function below will be used for optimization.

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

In the equation above the first term is the average squared-error(MSE) between the network output and wanted result across training samples. The second term is a regularization term which is Tykhonov regularization with parameter λ. Finally the third term is Kullback-Leibler divergence (KL divergence). This term help to make the hidden unit activation sparse. Weight of KL divergence is controlled with β while the sparsity level is tuned with ρ in a bernoulli distribution with mean ρ and another with mean $\tilde{\rho_b}$, which is the mean activation of b.

1.1 Part a

In this part preprocessing of the data has been completed. In this step firstly 16x16 RGB images are read and they converted to grayscale using luminosity model, which is given below.

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

Then we are asked to normalize the data. To realize that firstly means of each image are subtracted from themselves. Then standard deviation is calculated and the data are clipped at range $[-3 * std, 3 * std]$. Then normalization formula is applied and after that the data is mapped into [0.1,0.9] range,(0.8*Normalized+0.1).The normalization formula is given below. Also you can see the random 200 images before pre-processing and after pre-processing in fig.1 and fig.2.

$$Normalized = \frac{x - min(x)}{max(x) - min(x)}$$

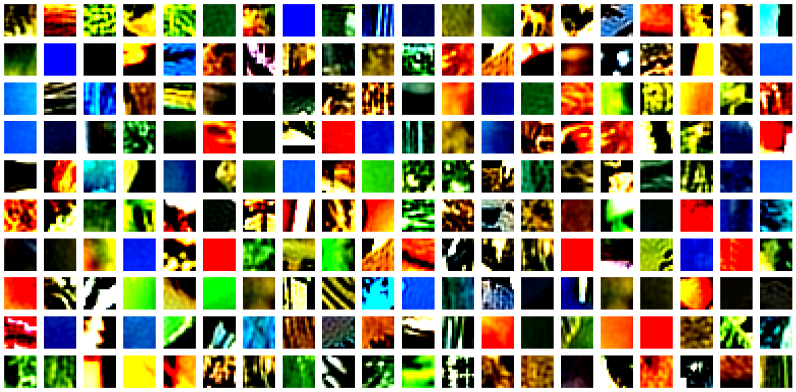


Figure 1: 200 random images before processing

Figure 2: 200 random images after processing

When we examine the figure 1 and figure 2, it can be clearly seen that most of the gray scale images are quite similar to their original versions and they are just gray versions of those images. I mean the color differences can be understood from the different shades. However, some of the grayscale images are slightly different than their original version. The reason behind this discrepancy is that we have clipped the data. However, in any case we are now ready to extract the features.

1.2 Part b

In this part we are firstly going to initialize weights and bias in the interval $[-\omega_o, \omega_o]$ where $\omega_o = \sqrt{\frac{6}{L_{pre}+L_{post}}}$, then a cost function which is asCost will be written. This function calculates the cost and the partial derivatives of weights and biases.The outputs of aeCost will be J and J_{grad}. Those terms will be given to gradient descent solver to minimize the cost that we have defined before after define and calculate parameters this solver is just an classical neural network architecture, with activation function sigmoid ($\sigma(x) = \frac{e^x}{1+e^x}$). The all equations and partial derivatives are given below for the cost function.

$$W_{all} = \begin{bmatrix} W \\ \beta \end{bmatrix}$$

1.2.1 MSE

$$MSE = \frac{1}{2N} \sum_{i=1}^{N} \|X - \tilde{X}\|^2$$

$$Y = \sigma(\sigma(\tilde{X} * W_{all_{hidden}}))$$

$$\delta_{out} = -\sigma'(V_{out}) \odot (X - \tilde{X})$$
$$\delta_{hid} = \delta'(V_{hid}) \odot W_{out}^T \delta_{out}$$

Partial derivatives are

$$\frac{\partial}{\partial w_{out}} MSE = W_{hid}^T \delta_{out}$$
$$\frac{\partial}{\partial w_{hid}} MSE = W_{out}^T \delta_{hid}$$
$$\frac{\partial}{\partial b_{out}} MSE = \frac{1}{N} \sum \delta_{out}$$
$$\frac{\partial}{\partial b_{hid}} MSE = \frac{1}{N} \sum \delta_{hid}$$

1.2.2 Tykhonov Regularization

Tykhonov Regularization=TRL

$$TRL = \frac{\lambda}{2} \left[\sum_{b=1}^{L_{hid}} \sum_{a=1}^{L_{in}} (W_{a,b}^{(1)})^2 + \sum_{c=1}^{L_{out}} \sum_{b=1}^{L_{hid}} (W_{b,c}^{(2)})^2 \right]$$

Partial derivatives are

$$\frac{\partial}{\partial w_{out}} TRL = \lambda W^{(2)}$$
$$\frac{\partial}{\partial w_{hid}} TRL = \lambda W^{(1)}$$
$$\frac{\partial}{\partial b_{out}} TRL = 0$$
$$\frac{\partial}{\partial b_{hid}} TRL = 0$$

1.2.3 Kullback-Leibler

Kullback-Leibler=KL

$$\beta \sum_{b=1}^{L_{hid}} KL(\rho|\tilde{\rho}_b)$$

Given that the distribution is bernoulli with parameter ρ given $\tilde{\rho}$, so the corresponding equation is given below.

$$\tilde{\rho} = \frac{1}{N} \sum \sigma(X \odot W_{hid} + b_{hid})$$

$$KL = \beta \sum_{b=1}^{L_{hid}} \rho \log\left(\frac{\rho}{\tilde{\rho}}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\tilde{\rho}}\right)$$

Partial derivatives are

$$\frac{\partial}{\partial w_{out}} KL = 0$$

$$\frac{\partial}{\partial w_{hid}} KL = \beta\left[\frac{1-\rho}{1-\tilde{\rho}} - \frac{\rho}{\tilde{\rho}}\right]$$

$$\frac{\partial}{\partial b_{out}} KL = 0$$

$$\frac{\partial}{\partial b_{hid}} KL = 0$$

Notice that $\frac{\partial}{\partial w_{hid}} KL$ is a column vector and the operations are done according to this.

1.2.4 Summation

Known that $J_{ae} = MSE + KL + TRL$; therefore we also have the equation below.

$$\frac{\partial}{\partial x} J_{ae} = \frac{\partial}{\partial x} MSE + \frac{\partial}{\partial x} KL + \frac{\partial}{\partial x} TRL$$

This equation gives:

$$\frac{\partial}{\partial w_{out}} J_{ae} = W_{hid}^T \delta_{out} + \lambda W^{(2)}$$

$$\frac{\partial}{\partial w_{hid}} J_{ae} = W_{out}^T \delta_{hid} + \lambda W^{(1)} + \beta\left[\frac{1-\rho}{1-\tilde{\rho}} - \frac{\rho}{\tilde{\rho}}\right]$$

$$\frac{\partial}{\partial b_{out}} J_{ae} = \frac{1}{N} \sum \delta_{out}$$

$$\frac{\partial}{\partial b_{hid}} J_{ae} = \frac{1}{N} \sum \delta_{hid}$$

Those equations are used in aeCost function, you can check the function from appendix.

1.3 Part c

In this part we are asked to obtain the optimized weights, and display the first layer of connection weights for each hidden neuron as separate images.Those images can be seen below.

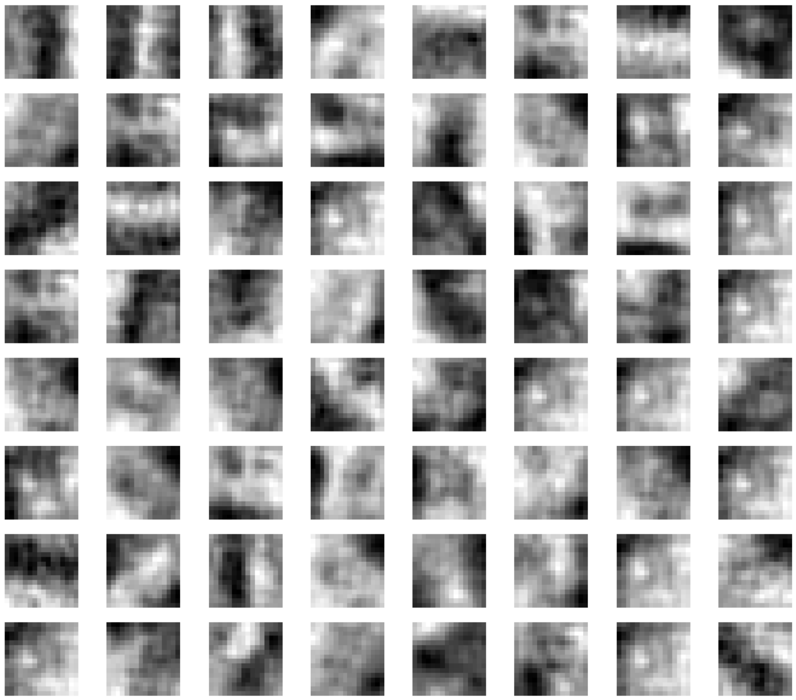


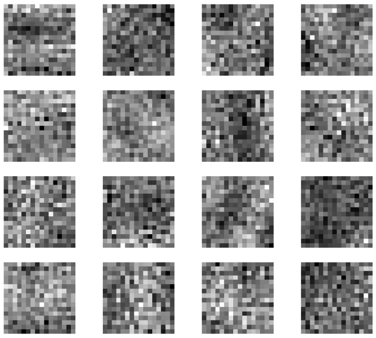
Figure 3: Visualization of First Layer of Connection Weights

When the figure above is examined it can be said that those images are look like the small parts of big images. I mean there are lots of line-like or curve-like with different orientations can be observed from those weights. Therefore, it can be said that those images are not the variations of the original data, they are the parts that create the original images. Namely those are the features of the data.

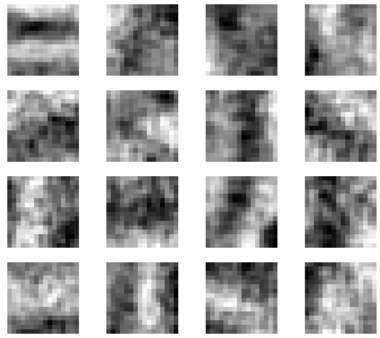
1.4 Part d

In this section we are asked to retrain the network for different values. In this training we are needed to choose 3 different $L_{hid} \in [10, 100]$ and 3 different $\lambda \in [0, 10^{-3}]$. Therefore, in total we are going to train this network for 9 more times for those parameters. The results of those trains can be seen below.

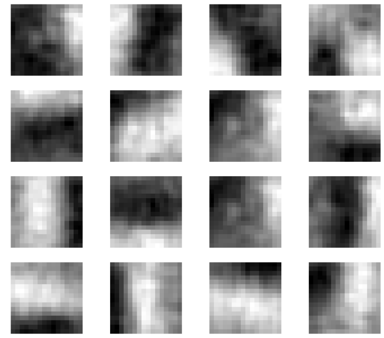
In figure 4, 5, 6 we can observe those 9 different cases. As you can see L_{hid} is the number of images. Also it can be said that the number of hidden layers affects the overall learning process. This is because, when it is increased, we can observe more different and complex plots in those figures.Actually number of those neurons are related to network capacity. However, the increase of network capacity does not mean better results every time, it can lead to overfitting in some cases since it starts to learn complex details. On the other hand,



(a) $L_{hid} = 16$, $\lambda = 0$



(b) $L_{hid} = 16$, $\lambda = 2 * 10^{-4}$

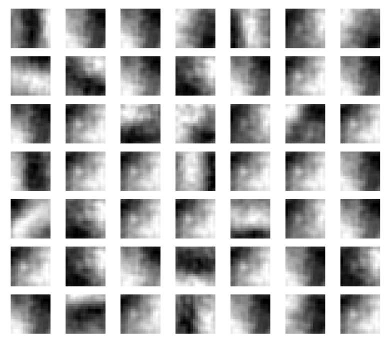


(c) $L_{hid} = 16$, $\lambda = 10^{-3}$

Figure 4: Images for $L_{hid} = 16$

(a) $L_{hid} = 49$, $\lambda = 0$

(b) $L_{hid} = 49$, $\lambda = 2 * 10^{-4}$

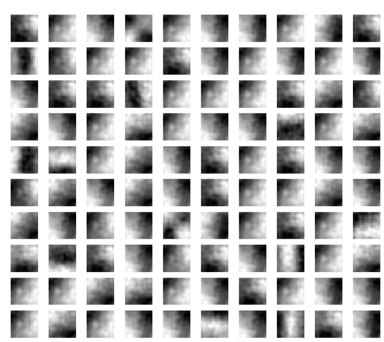


(c) $L_{hid} = 49$, $\lambda = 10^{-3}$

Figure 5: Images for $L_{hid} = 49$

(a) $L_{hid} = 100$, $\lambda = 0$

(b) $L_{hid} = 100$, $\lambda = 2 * 10^{-4}$



(c) $L_{hid} = 100$, $\lambda = 10^{-3}$

Figure 6: Images for $L_{hid} = 100$

small number of neurons may not be enough to learn all necessary features. Therefore, medium number can be said to be best.

Additionally in figure 4,5,6 we can examine the images for different λ values. As you can see when λ is higher the images are smoother. It can be said that λ helps the network in learning process by preventing it from overfitting. In this way the networks learn better rather than memorize the original data. However, as we can see from the equation, λ is in the regularization term. Therefore, we need to arrange the value of λ according to this terms, since if this term is too large then the cost will be bigger and it will corrupt the learning. Therefore, we should pick a medium λ.

2 Question 2

In the second question, we considered a model for examining sequences of words. The task was to get a prediction of the fourth word from the given 3 preceding words (Trigram).The architecture of this network is shown below.

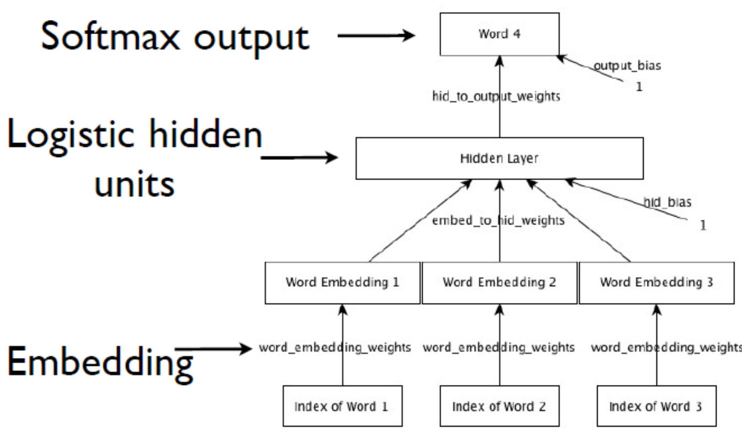


Figure 7: Network Architecture

As seen from the architecture we need to start to task from embedding layer. In order to give the inputs to this layer, indexes of the words in the given datasets has been converted to vector representations using one hot encoding method. The length of those vectors are the given dictionary size. Additionally in the question, it is indicated that the activation function of the hidden layer is sigmoid function while the output activation function is softmax function. ($Softmax : o_i = \frac{e^{z_i}}{\sum_{j=1}^{250} e^{z_j}}$, (250 neurons)), ($Sigmoid : \sigma(x) = \frac{e^x}{1+e^x}$)

2.1 Part a

In this part of the question we are required to implement stochastic gradient algorithm using the following parameters.

- Mini Batch Size=200

- Learning Rate (η)=0.15
- Momentum Rate= α=0.85
- Max Epoch=50
- Gaussian variables weights and biases with std= 0.01
- (D,P)=(32,256),(16,128),(8,64)

In order to implement the gradient descent I firstly needed to give the one hot encoded indices to embedding layer. In order to continue after this process I thought the embedding layer with an activation. To avoid any error the activation is g(x)=x, which is linear activation. Therefore, in total I have three different activation function. Therefore, I have added act_f function to select proper activation. Additionally, I had to add a function to choose the neuron, and a function to get the derivatives of activations. The derivatives are given below.

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

The forward propagations of the network is quite standard, it gives the outputs of the activation functions to the next layer. The backpropagation is the most complex part of the question. In the backpropagation part δ is calculated like question 1, then update parameter is calculated. Using this update variable and momentum new weights are calculated.

$$\delta^(t) = (act)' * ((W^(t+1))^T \odot \delta^(t+1))$$

Where (act)' represents the derivative of last activation function.

$$UP^{(t)} = \eta * \delta \odot V$$

$$W_{all} = W_{all} + \frac{UP}{batchsize} + \alpha * \frac{UP^{(t-1)}}{batchsize}$$

As a loss function I have used cross entropy error, in the train function. The train function is a classical train function of the neural network codes.

$$L(X,\tilde{X}) = \frac{-1}{N}\sum_{j=1}^{N}\sum_{i=1}^{M} x_{j,i}\log(x_{pred,j,i}) = \frac{-1}{N}\sum_{j=1}^{N}\tilde{X} * \log\big(X^T\big)$$

While the loss function is Cross Entropy loss the delta is given below.

$$\delta_{out} = X - \tilde{X}$$

The summary of the logic of the code is over, all the codes can be found in appendix part.

2.1.1 Optimization

I tried the codes with different parameters for several times and finally I have choose the parameters below.

- Learning Rate (η)=0.25
- Momentum Rate= α=0.8
- Gaussian variables weights and biases with std= 0.25

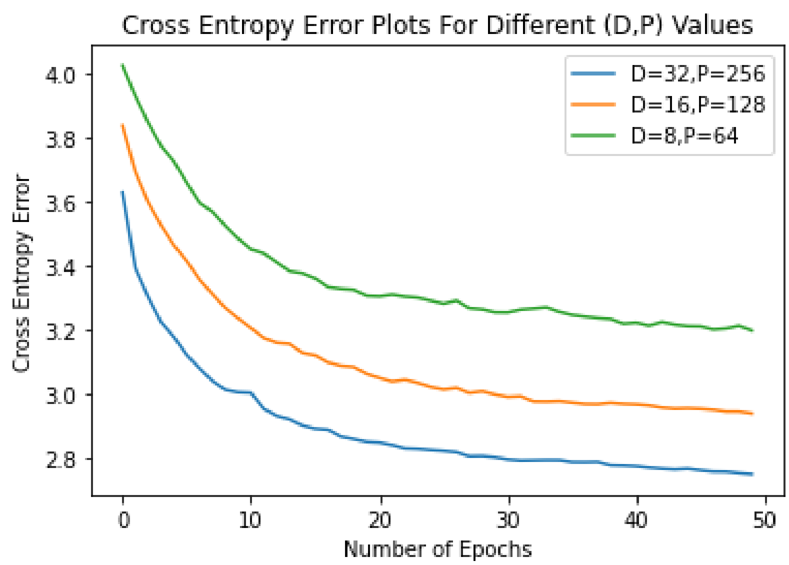


Figure 8: Cross Entropy Error curves for 3 different (D,P)

As seen from the figure, as expected loss is decreasing with epochs.However, after some points, losses converge.Additionally, with the decrease of (D,P) the error becomes bigger. The reason for this increase in error is that when D decreases mapping of words become more lossy. Also, when the hidden neuron number decreases learning capacity of the model decreases which is not a desired case as we know from previous question.
Additionally for this report I have not used early stopping based on cross entropy but in the code I have uploaded there is an early stopping. If the cross entropy loss is not changed more than 0.5% then it stops.

2.2 Part B

In this part of the question we are asked to predict 10 most possible words of the 4th word of 5 different Trigram. For this purpose I have written a predict function which is quite straingforward. It calculate forward propagation then sort and takes the highest 10 probability outputs.

I have used the predict function to create predictions for 3 of (D,P). You can see the results below.

```
Triagram 1:

do nt take

Most Probable 10 Predictions are:

many
street
can
here
by
:
big
including
of
house
```

(a) Trigram 1

```
Triagram 2:

going to see

Most Probable 10 Predictions are:

two
will
american
big
many
,
can
an
john
at
```

(b) Trigram 2

```
Triagram 3:

the old days

Most Probable 10 Predictions are:

here
,
john
take
see
will
would
which
's
big
```

(c) Trigram 3

```
Triagram 4:

it was nt

Most Probable 10 Predictions are:

an
american
few
ms.
part
too
see
second
members
and
```

(d) Trigram 4

```
Triagram 5:

do they like

Most Probable 10 Predictions are:

members
how
of
federal
among
right
can
under
,
big
```

(e) Trigram 5

Figure 9: Most probable 10 predictions for 5 Trigram when (D,P)=(32,256)

```
Triagram 1:

do nt take

Most Probable 10 Predictions are:

here
an
,
members
street
second
with
if
used
can
```

(a) Trigram 1

```
Triagram 2:

going to see

Most Probable 10 Predictions are:

many
,
members
will
street
right
big
what
american
among
```

(b) Trigram 2

```
Triagram 3:

the old days

Most Probable 10 Predictions are:

american
,
here
big
will
if
members
's
for
court
```

(c) Trigram 3

```
Triagram 4:

it was nt

Most Probable 10 Predictions are:

here
an
,
members
would
states
second
night
by
house
```

(d) Trigram 4

```
Triagram 5:

do they like

Most Probable 10 Predictions are:

here
members
more
,
big
take
will
police
because
ms.
```

(e) Trigram 5

Figure 10: Most probable 10 predictions for 5 Trigram when (D,P)=(16,128)

```
Triagram 1:

do nt take

Most Probable 10 Predictions are:

here
more
big
,
not
right
members
by
under
has
```

(a) Trigram 1

```
Triagram 2:

going to see

Most Probable 10 Predictions are:

,
here
american
will
how
members
big
many
street
's
```

(b) Trigram 2

```
Triagram 3:

the old days

Most Probable 10 Predictions are:

here
,
american
big
members
by
will
how
's
for
```

(c) Trigram 3

```
Triagram 4:

it was nt

Most Probable 10 Predictions are:

here
an
john
part
,
former
how
would
many
house
```

(d) Trigram 4

```
Triagram 5:

do they like

Most Probable 10 Predictions are:

here
more
an
,
big
members
ms.
are
day
by
```

(e) Trigram 5

Figure 11: Most probable 10 predictions for 5 Trigram when (D,P)=(8,64)

As seen from the figure 9,10,11 we get 10 predictions for each 5 trigrams. While (D,P)=(32,256) has more meaningful predictions like "going to see two ..." (D,P)=(8,64) predictions are mostly have no meanings. It was expected from previous part and by figure 8.

3 Question 3

In this question we are going to create a network that will classify the human activities (downstairs=1, jogging=2, sitting=3, standing=4, upstairs=5, walking=6) according to movement signals, those signals are obtained from three different sensors for 150 time units. We have two different data-sets, which are training set with 3000 samples and test set with 600 samples.

3.1 Part A

Firstly, we will create a single layered RNN with hidden 128 neurons ,using the back propagation through time. In this process hyperbolic tangent (tanh) activation function. Then it will be followed by multi-layer perceptron network with softmax activation function will be used for classification. Throughout this processes I will use a stochastic gradient descent algorithm with learning rate ($\eta = 0.1$),momentum rate($\alpha = 0.85$) and maximum 50 epochs while the distibution of the weights and biases are Xavier Uniform distribution, which is given below.

$$W = [-a, a]$$
$$a = \sqrt{\frac{6}{L_{prev} + L_{next}}}$$

Then layer class initialization is similar to question 2, but the distribution for weights are different. For act_f function tanh and its derivative are added. Also a new function ($rec_a ct$) is added to lyr class, because now we are dealing with time series problem and we need to use previous sample as well.

After layer class, I have created RNN class, which contains forward propagation, back propagation, train, etc. The main idea and equation of the RNN network forward propagation is given below.

$$h(t) = f_H(W_{IH}x(t) + W_{HH}h(t-1))$$
$$y(t = f_o(W_{HO}h(t)))$$

The forward propagation function is divided to two parts which are that one for the recurrent layer and the other one is for the MLP layers. For the recurrent layer, it starts at time unit 0 and continue till the end time. At each iteration, the $rec_a ct$ function is used and the current value is updated and its value is assigned for next prevupdate. The forward propagation of the MLP layers is the classical one like the previous questions, but differently

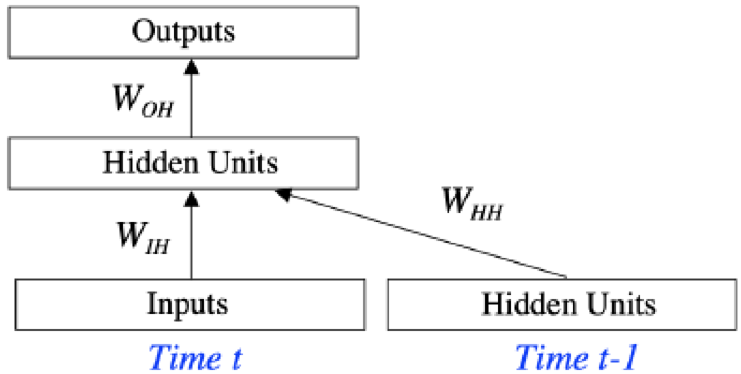


Figure 12: RNN Schematic

they use ReLU activation function.

For back propagation, both MLP back propagation and Backpropagation Through Time (BPTT) are used. Then the combination of those are used for update of weights.

$$W = W - (\alpha * W_{moment} + \eta * W_{up})$$

$$W_{up} = \sum_{t}^{T} \frac{\partial E(t)}{\partial W}$$

Similarly W_{moment} is updated at each epoch. The full backpropagation can be seen in back function from RNN class.
The other elements of the RNN class are classical functions like before. However, there are two different functions which are predict anc conf. Predict function brings the accuracy of the prediction, conf function creates the confusion matrix of the predictions over the testset.The results of the network are given below in figure 13 and 14.

As it can be seen from figure 13 and 14 the results are not good. Firstly it is obvious that the loss plot is not stable.Namely, we cannot see any converge behaviour in the loss. Also from confusion matrices it can be seen that the network is prone to predict a certain behaviour. This instability is come from the gradients of the weights since they become very high or low during learning process. This can stop the learning of algorithm or it can cause the program to get warnings like Nan outputs.The reason for gradient explosion is probable due to the cumulative effect of Backpropagation Through Time (BPTT). This algorithm is used to train recurrent neural networks. It works by propagating the error backwards through time. This can lead to the gradients to accumulate to extremely high values, which can lead to instability.There are several potential solutions for this problem.Reducing the learning rate can help. It will slow the accumulation of weights, and make the network less vulnerable.Using clipping of gradients can help. It will limit the size of the gradients, and prevent them to become too large.Using normalization of layers can help . These layers can normalize the gradients and map them to a specific range. However, those solutions can be

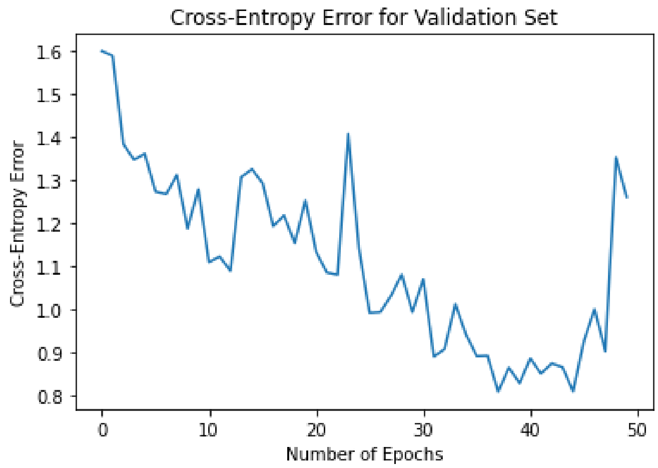
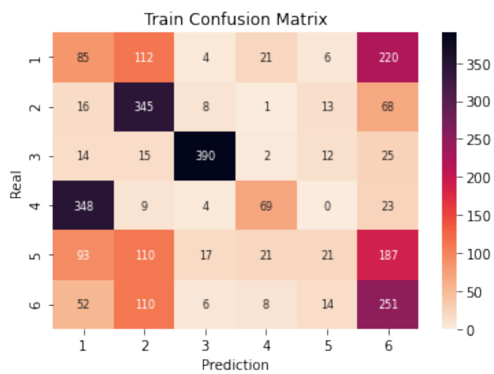
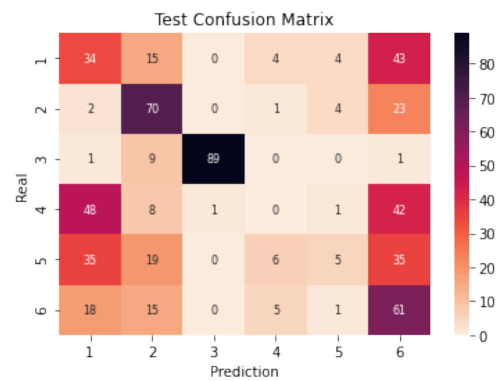


Figure 13: Cross Entropy Loss per Epoch for 50 epochs for RNN



(a) Confusion Matrix for Train,Train Accuracy= 43.0%



(b) Confusion Matrix for Test, Test Accuracy= 43.164%

Figure 14: Confusion Matrices

used together, or also there is possibility that none of them works.

3.2 Part B

In this part we are going to repeat the previous part but rather than RNN we are going to use Long Short Term Memory(LSTM). This algorithm is better to handle the recurrent layers problems like extremely high or low gradients. In figure 15 you can see an LSTM cell. As seen from figure there are several gates in this structure and the equations for those gates are given below.

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$
$$c_t = tanh(W_v.[h_{t-1}, x_t] + b_v)$$

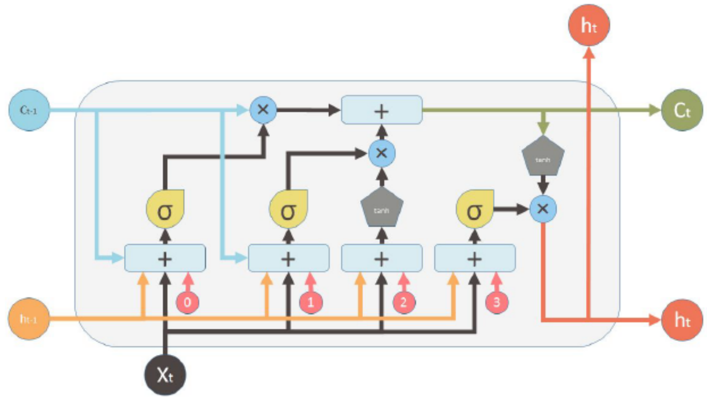


Figure 15: Schematic of LSTM cell

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$
$$C_{memory,t} = f_t * C_{memory,t-1} + i_t * c_t$$
$$h_t = o_t * tanh(C_{memory,t})$$

The backpropagation of the LSTM is similar to BPTT of RNN. The full implementation of the BPTT of LSTM can be seen from back function in the LSTM class. The rest of the LSTM class is also almost same with the RNN class.

After implementing the LSTM network, I got the results below.

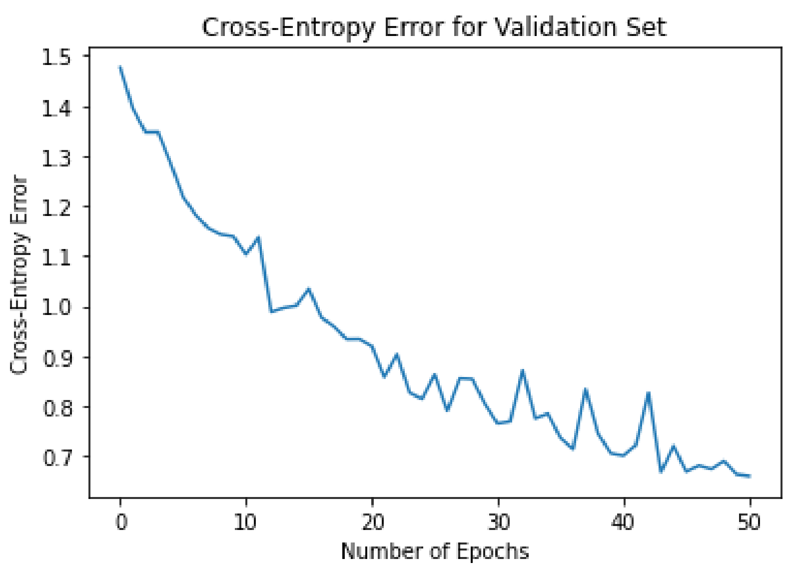
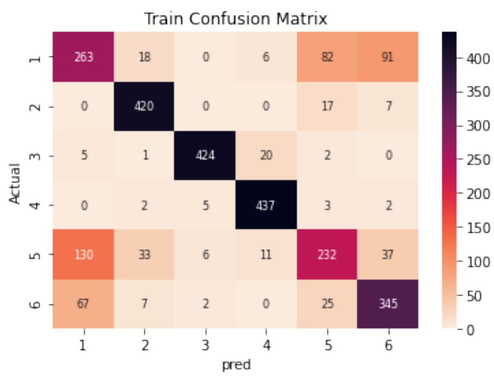
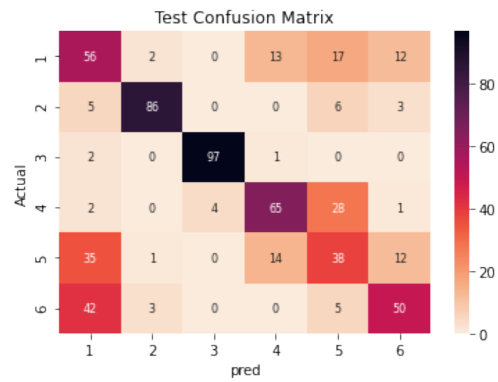


Figure 16: Cross Entropy Loss per Epoch for 50 epochs for LSTM

As we can see from figure 16 and 17 LSTM brought a big improvement to RNN. When we observe the figures, we can easily see that while the losses becomes more stable and the losses



(a) Confusion Matrix for Train,Train Accuracy: 78.555%



(b) Confusion Matrix for Test,Test Accuracy: 65.33%

Figure 17: Confusion Matrices

seems more prone to converge, the accuracy becomes better. I got an 78.55% train accuracy and 65.33% test accuracy. Even if those are not perfect, they are still huge improvements because they were respectively 43% and 43.164% in RNN results. Also from confusion matrices it can be seen that there is no single prediction problem anymore. Therefore, it can be said that the network can learn better.

4 Part C

In this part we are going to repeat the previous part but rather than RNN or LSTM we are going to use Gated Recurrent Unit, which is an cell-based network like LSTM.Gru has less gates than LSTM. In figure 18 you can see an LSTM cell. As seen from figure there are several gates in this structure and the equations for those gates are given below.

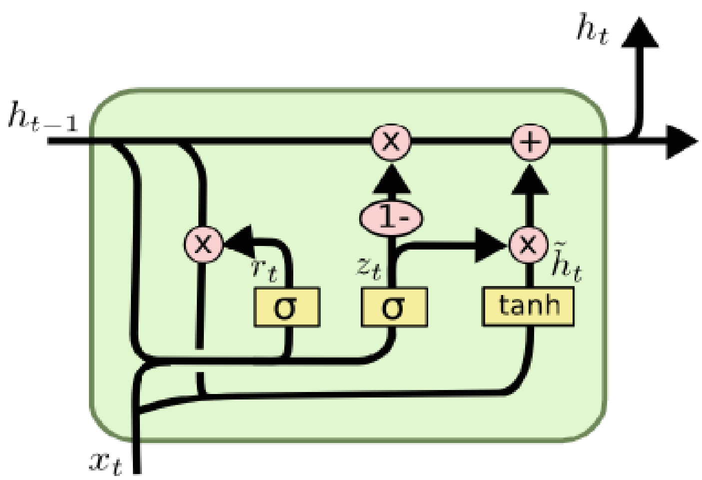


Figure 18: Schematic of GRU cell

$$z_t = \sigma(x_t.W_z + U_z.ht - 1 + b_z)$$
$$r_t = \sigma(x_t.W_r + U_r.ht - 1 + b_r)$$
$$\tilde{h}(t) = tanh(x_t.W_h + U_h.ht - 1 * r_t + b_h)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}(t)$$

The backpropagation of the GRU is similar to BPTT of LSTM. The full implementation of the BPTT of GRU can be seen from back function in the GRU class. The rest of the GRU class is also almost same with the LSTM class.

After implementing the GRU network, I got the results below.

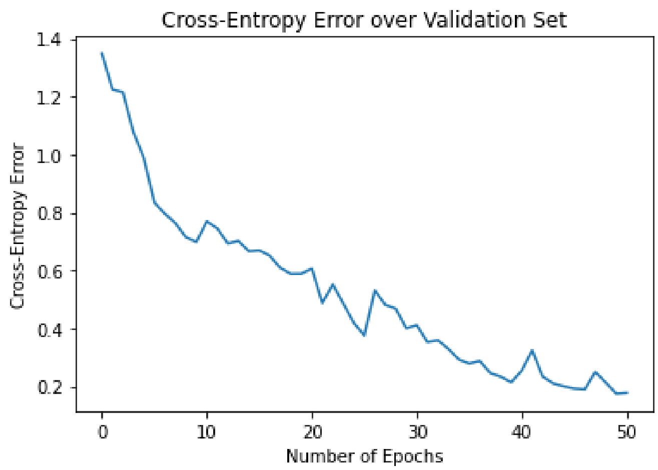
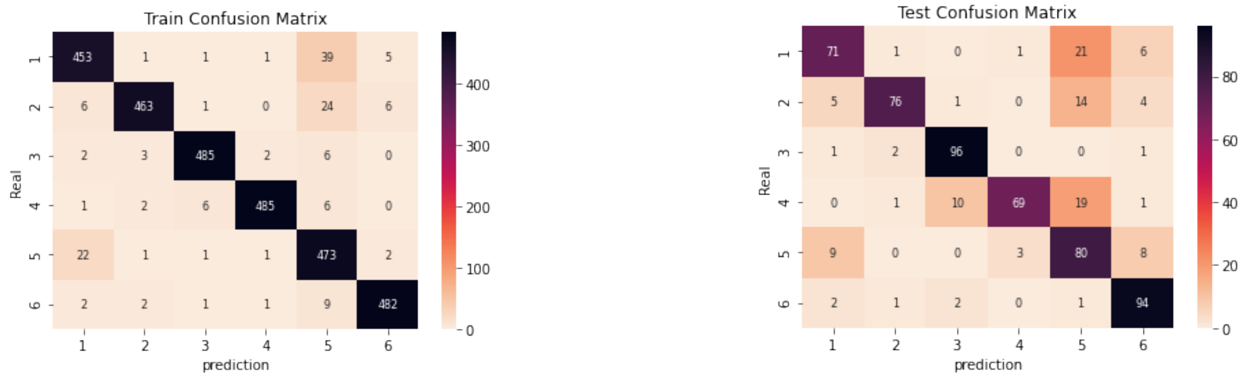


Figure 19: Cross Entropy Loss per Epoch for 50 epochs for GRU

As we can see from figure 19 and 20 gru brought a big improvement to LSTM. When we observe the figures, we can easily see that the losses seems more prone to converge, the accuracy becomes better. I got an 94.85% train accuracy and 81% test accuracy. Even if those are not perfect, they are still huge improvements because they were respectively 78.55% and 65.33% in LSTM results. Also from confusion matrices it can be seen that the orientations of model to predict some behaviours changed and its much more better now.Also it is observed that GRU is faster than LSTM.
Even if GRU bring better results LSTM is more stable than GRU.In addition to that there is a high error possibility when trying to work with high learning rates.Therefore, I made the learning rate 0.3 for GRU.

When comparing all the results it can be said that RNN is the worst among those 3 models.If we want the training to be time efficient or if there are low time units GRU can be chosen, it there are high time units LSTM can be used. Since it has less vulnerable to overflows and can be more accurate with higher learning rates and high time units.



(a) Confusion Matrix for Train,Train Accuracy: 94.85%

(b) Confusion Matrix for Test,Test Accuracy: 81%

Figure 20: Confusion Matrices

5 Appendix- Codes

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
import h5py
import os
import sys
import warnings
warnings.filterwarnings("ignore")

def initialize_parameters(Lin, Lhid):

    np.random.seed(443)
    Lout=Lin
    w_0 =np.sqrt(6/(Lin+Lhid))
    w_1=np.sqrt(6/(Lhid+Lout))

    W1 = np.random.uniform(-w_0,w_0,(Lin,Lhid))
    b1 = np.random.uniform(-w_0,w_0,(1,Lhid))


    W2 = W1.T # at the same time W2=W1.T
    b2 = np.random.uniform(-w_1,w_1,(1,Lout))
    return (W1, W2, b1, b2), (0,0,0,0)

sigmoid = lambda x:np.exp(x)/ (1+np.exp(x))
derSigmoid= lambda x: sigmoid(x)*(1-sigmoid(x))

def forward(W_e, data):
```

```python
    W1, W2, b1, b2 = W_e

    W1_ = data.dot(W1)+b1
    z = sigmoid(W1_)
    dz= derSigmoid(W1_)


    W2_ = z.dot(W2)+b2
    z2 = sigmoid(W2_)
    dz2= derSigmoid(W2_)

    cac=(data,dz,dz2)

    return z, z2,cac

def aeCost(W_e, data, params):
    (Lin, Lhid, lmb, beta, rho) = params
    W1, W2, b1, b2 = W_e
    N= len(data)

    hid, out, (_,d_hid,d_out) = forward(W_e, data)
    hid_mean = hid.mean(axis=0,keepdims=True)

    ASE = (1/(2*N)) * np.sum(np.power((data - out),2)) #mse

    TYK = (lmb/2) * (np.sum(W1**2) + np.sum(W2**2)) # Tykhonov
    KL = rho*np.log(rho/hid_mean) + (1-rho)*np.log((1-rho)/(1-hid_mean))
    KL =  beta * KL.sum() # Kullback-Leibler

    J = ASE + TYK + KL ## J_ae

    d_ASE=-(data-out)/N
    d_TYK=W1*lmb , W2*lmb
    d_KL=beta*(- rho/hid_mean + (1-rho)/(1 - hid_mean))/N

    d1 = d_ASE * d_out
    dW2 = hid.T.dot(d1) + d_TYK[1]
    db2 = d1.sum(axis=0, keepdims=True)

    d2 = d_hid * (d1 .dot( W2.T) + d_KL)

    dW1 = data.T .dot(d2) + d_TYK[0]
    db1 = d2.sum(axis=0, keepdims=True)


    J_grad=(dW1,dW2,db1,db2)
    return J, J_grad
```

```python

def update_parameters(We, mW, dW, m, l_rate):
    mW = m * np.array(dW, dtype=object) + l_rate * np.array(mW, dtype=object)

    We = np.array(We, dtype=object) - mW

    We = tuple(We)
    mW = tuple(mW)

    return We, mW

def solver( data, params, eta, alpha, epoch, batch):

    loss_list = []
    if batch is None:
        batch = len(data)

    Lin = params[0]
    Lhid = params[1]

    We, mWe = initialize_parameters(Lin,  Lhid)

    iter_ep = int(len(data) / batch)
    # j_prev=0  ##early stopping but not used
    for k in range(epoch):
        J_sum = 0

        temp_start=0
        temp_end = batch



        for j in range(iter_ep):

            data_temp = data[temp_start:temp_end]

            J, Jgr = aeCost(We, data_temp, params)



            We, mWe = update_parameters(We, mWe, Jgr, eta, alpha)

            temp_start = temp_end
            temp_end += batch
            J_sum += J

        J_sum = J_sum/iter_ep
```

```
        #earl=abs(J_sum-j_prev)/abs(J_sum) ## early stopping but not used
        #j_prev=J_sum ##  early stopping but not used

        #if earl<0.0001:
        #    break

        print("Loss: {:.2f} [Epoch {} of {}]".format(J_sum, k+1, epoch))
        loss_list.append(J_sum)

    return We, loss_list

def plot_weights(we):
    (w1,w2,b1,b2) = we
    fig=plt.figure(figsize=(18, 16))
    plot_shape = int(np.sqrt(w1.shape[1]))
    for i in range(w1.shape[1]):
        plt.subplot(plot_shape,plot_shape,i+1)
        plt.imshow(np.reshape(w1[:,i],(16,16)), cmap='gray')
        plt.axis('off')
    plt.show()

def label_1h(y, size):
    output = np.zeros((len(y), size))
    for q in range(len(y)):
        temp = np.zeros(size)
        temp[y[q]-1] = 1
        output[q,:] = temp
    return output

def data_1h(x, size):
    output = np.zeros((len(x), x.shape[1], size))
    for q in range(len(x)):
        for p in range(x.shape[1]):
            temp = np.zeros(size)
            temp[x[q,p]-1] = 1
            output[q,p,:] = temp
    return output

class Layer:
    def __init__(self,dim_in,neur_N,avg,std, act):
        self.dim_in = dim_in
        self.neur_N = neur_N
        self.act = act
        self.prevU = 0
        self.Last_Act=None
        self.err_layer=None
        self.delta_layer=None
```

```
        if self.act == 'softmax' or self.act == 'sigmoid':
            self.all_W= np.random.normal(avg,std, (neur_N,dim_in+1))
            self.weight=self.all_W[:,:-1]
            self.bias=self.all_W[:,-1:]
        else:
            self.D = dim_in
            self.S_dict = neur_N
            self.weight = np.random.normal(avg, std, (self.S_dict,self.D))

    def act_f(self, x):

        if(self.act == 'softmax'):
            e_x = np.exp(x - np.max(x))
            return e_x/np.sum(e_x, axis=0)

        elif(self.act == 'sigmoid'):
            return np.exp(2*x)/(1+np.exp(2*x))
        else:
            return x

    def act_N(self,x):
        if self.act == 'sigmoid' or self.act == 'softmax':
            samp_N = x.shape[1]
            inp_temp = np.r_[x, [np.ones(samp_N)*-1]]
            self.Last_Act = self.act_f(self.all_W.dot(inp_temp))

        else:
            Embed = np.zeros((x.shape[0],x.shape[1], self.D))
            for m in range(Embed.shape[0]):
                Embed[m,:,:] = self.act_f(x[m,:,:].dot(self.weight))
            Embed = Embed.reshape((Embed.shape[0], Embed.shape[1] *
            ↪ Embed.shape[2]))
            self.Last_Act = Embed.T
        return self.Last_Act

    def act_der(self, x):
        if(self.act == 'sigmoid' or self.act == 'softmax'):
            return x*(1-x)
        else:
            return np.ones(x.shape)




class Neural:
    def __init__(self):
        self.lyrs=[]
```

```
    def layer_add(self,layer):
        self.lyrs.append(layer)

    def Forward(self,data_train):
        IN=data_train
        for layer in self.lyrs:
            IN=layer.act_N(IN)
        return IN

    def Back(self,l_rate,size_batch,data_train,label_train,momentum):
        out_forwa = self.Forward(data_train)
        for i in reversed(range(len(self.lyrs))):
            lyr = self.lyrs[i]

            if(lyr == self.lyrs[-1]):
                lyr.delta_layer=label_train.T-out_forwa
            else:
                layer_next = self.lyrs[i+1]
                lyr.err_layer = np.matmul(layer_next.weight.T, layer_next.delta_layer)
                der=lyr.act_der(lyr.Last_Act)
                lyr.delta_layer=der*lyr.err_layer

        for i in range(len(self.lyrs)):
            lyr = self.lyrs[i]
            if(i == 0):
                inp_temp = data_train
            else:
                samp_N = self.lyrs[i - 1].Last_Act.shape[1]
                inp_temp = np.r_[self.lyrs[i - 1].Last_Act, [np.ones(samp_N)*-1]]


            if(lyr.act == 'sigmoid' or lyr.act == 'softmax'):
                update =  l_rate*np.matmul(lyr.delta_layer, inp_temp.T)
                lyr.all_W+= update/size_batch + (momentum*lyr.prevU)
            else:
                emb_delta = lyr.delta_layer.reshape((3,size_batch,lyr.D))
                inp_temp = np.transpose(inp_temp, (1,0,2))
                update = np.zeros((inp_temp.shape[2], emb_delta.shape[2]))
                for i in range(emb_delta.shape[0]):
                    update += l_rate * np.matmul(inp_temp[i,:,:].T, emb_delta[i,:,:])
                update = update
                lyr.weight += update/size_batch + (momentum*lyr.prevU)
            lyr.prevU = update/size_batch

```

```python
    def Train(self,l_rate,size_batch,data_train,label_train, data_test, lbl_test, num_ep,momentum,S_dict):
        losses = []
        temp_loss=0
        for ep in range(num_ep):

            print("Ep:",ep)
            indexing=np.random.permutation(len(data_train))
            data_train=data_train[indexing,:]
            label_train=label_train[indexing]
            batch_N = int(np.floor(len(data_train)/size_batch))
            for j in range(batch_N):
                data_one_hot = data_1h(data_train[j*size_batch:size_batch*(j+1),:], S_dict)
                label_one_hot = label_1h(label_train[j*size_batch:size_batch*(j+1)], S_dict)
                self.Back(l_rate,size_batch,data_one_hot,label_one_hot,momentum)

            out_val = self.Forward(data_test)
            los_cros = - np.sum(np.log(out_val) * lbl_test.T)/out_val.shape[1]
            print('C-E Error ', los_cros)
            losses.append(los_cros)
            print(abs(los_cros-temp_loss)/abs(los_cros))
            if abs(los_cros-temp_loss)/abs(los_cros)<0.005:
                print("stopped based on cross-entropy")
                break
            temp_loss=los_cros

        return losses


    def predict(self, inputIMG, k):
        out = self.Forward(inputIMG)
        return np.argsort(out, axis=0)[:,0:k]


class lyr: #lyr Class for RNN
    def __init__(self,dim_in,neur_N,act,beta):
        self.dim_in = dim_in
        self.neur_N = neur_N
        self.act = act
        self.beta=beta
        self.prevU = 0
        self.prevU_RNN = 0
        self.Last_Act=None
        self.err_lyr=None
        self.delta_lyr=None
```

```
        self.XUD=np.sqrt(6/(dim_in+neur_N))
        self.all_W =np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))
        self.W2=np.random.uniform(-self.XUD,self.XUD,(neur_N,neur_N))




    def act_f(self, x):

        if(self.act == 'softmax'):
            e_x = np.exp(x - np.max(x))
            return e_x/np.sum(e_x, axis=1, keepdims=True)
        elif(self.act == 'hyperbolic'):
            return np.tanh(x*self.beta)
        elif(self.act=="sigmoid"):
            return np.exp(2*x)/(1+np.exp(2*x))
        elif(self.act=="relu"):
            return np.maximum(0,x)
        else:
            return x

    def act_N(self,x):
        samp_N = len(x)
        inp_temp = np.concatenate((x, -np.ones((samp_N, 1))), axis=1)
        self.Last_Act = self.act_f(inp_temp.dot(self.all_W))
        return self.Last_Act

    def rec_Act(self,x,hid):
        samp_N = len(x)
        inp_temp = np.concatenate((x, -np.ones((samp_N, 1))), axis=1)
        last= hid.dot(self.W2)+ inp_temp.dot(self.all_W)
        self.Last_Act = self.act_f(last)
        return self.Last_Act

    def act_derv(self, x):
        if(self.act=="sigmoid" or self.act == 'softmax'):
            return x*(1-x)
        elif (self.act=="relu"):
            return (x>0)*1
        elif(self.act == 'hyperbolic'):
            return self.beta*(1-x*x)
        else:
            return np.ones(x.shape)

class RNN_:
    def __init__(self,data_train):
```

```
        self.samp_T= data_train.shape[1]
        self.del_rec=np.empty((32, self.samp_T, 128))
        self.loss_rec=np.empty((32, self.samp_T, 128))
        self.p_hid=np.zeros((32, 128))
        self.lyrs=[]

    def lyr_add(self,lyr):
        self.lyrs.append(lyr)

    def Forward(self,data_train):#Foward Prop
        samp_N, samp_T, D = data_train.shape
        inp=np.empty((samp_N, samp_T, 128))
        self.p_hid=np.zeros((samp_N, 128))
        for t in range(samp_T): #all time units are needed
            x = data_train[:, t]
            inp[:, t]=self.lyrs[0].rec_Act(x,self.p_hid)
            self.p_hid=inp[:, t]
        output = inp[:, -1]

        for lyr in self.lyrs[1:len(self.lyrs)]: # MLP
            output=lyr.act_N(output) #last  sample  contain memory
        return inp,output

    def back(self,l_rate,size_batch,data_train,lab_train,momentum):
        inp,output = self.Forward(data_train)
        out_forw = output
        for i in reversed(range(len(self.lyrs))):# backprop til recurrent
            lyr = self.lyrs[i]
            #out layer
            if(lyr == self.lyrs[-1]):
                lyr.delta_lyr=lab_train-out_forw
            else :
                layer_next = self.lyrs[i+1]
                lyr.err_lyr = layer_next.delta_lyr.dot(layer_next.all_W[:len(layer_next.all_W)-1].T)
                derv=lyr.act_derv(lyr.Last_Act)
                lyr.delta_lyr=derv*lyr.err_lyr
                if (lyr == self.lyrs[0]):
                    self.loss_rec[:,-1]=lyr.err_lyr
                    self.del_rec[:,-1]=lyr.delta_lyr

        d_all_weight=0
        d_hid_weight=0
        samp_N, samp_T, D = data_train.shape

        for t in reversed(range(samp_T)):
            lyr=self.lyrs[0]
```

```
            if t > 0:
                u = inp[:, t-1]
            else:
                u = np.zeros((samp_N, 128))

            derv=lyr.act_derv(u)
            d_hid_weight+=u.T.dot(self.del_rec[:,t])
            d_all_weight+=np.concatenate((data_train[:,t-1], -np.ones((len(data_train[:,t-1]), 1))), axis=1).T.dot(self.del_rec[:,t])

            # Recc dlt updte
            self.loss_rec[:,t-1]=self.del_rec[:,t].dot(lyr.W2.T)
            self.del_rec[:,t-1]=self.loss_rec[:,t-1]*derv


        for i in range(len(self.lyrs)): #update all weights
            lyr = self.lyrs[i]
            if(i == 0):
                lyr.prevU =  d_all_weight*l_rate/(150*size_batch)
                lyr.prevU_RNN =  d_hid_weight*l_rate/(150*size_batch)
                lyr.all_W+=  (momentum*lyr.prevU) +lyr.prevU
                lyr.W2+= (momentum*lyr.prevU_RNN) +lyr.prevU_RNN

            else:
                samp_N = len(self.lyrs[i - 1].Last_Act)
                inp_temp=np.concatenate((self.lyrs[i - 1].Last_Act, -1*np.ones((samp_N, 1))), axis=1)
                lyr.prevU =  inp_temp.T.dot(lyr.delta_lyr)*l_rate/size_batch
                lyr.all_W+= lyr.prevU*momentum +lyr.prevU

    def train(self,l_rate,size_batch,data_train,lab_train, inp_test, lab_test, N_ep,momentum):
        cr_los_list = []
        train_res=[]
        for ep in range(N_ep):
            print("ep:",ep)
            ind=np.random.permutation(len(data_train))
            data_train=data_train[ind]
            lab_train=lab_train[ind]
            batch_N = int(len(data_train)/size_batch)
            for j in range(batch_N):
                train_data = data_train[j*size_batch:(j+1)*size_batch]
                train_labels = lab_train[j*size_batch:(j+1)*size_batch]
                self.back(l_rate,size_batch,train_data,train_labels,momentum)
            _, out_val = self.Forward(inp_test)
            _, out_train = self.Forward(data_train)
```

```
            cr_los =  np.sum(-lab_test*np.log(out_val))/len(out_val)
            cr_los1 =  np.sum(-lab_train*np.log(out_train))/len(out_train)
            print('C-E Error of Validation', cr_los)
            print('C-E Error Error of Train', cr_los1)
            cr_los_list.append(cr_los)
            train_res.append(cr_los1)
        return cr_los_list, train_res

    def Predict(self,inps,realout):
        _,out = self.Forward(inps)
        out = out.argmax(axis=1)
        realout = realout.argmax(axis=1)
        return ((out == realout).mean()*100)

    def conf(self,inp,outp):
        _,pred= self.Forward(inp)
        pred = pred.argmax(axis=1)
        outp = outp.argmax(axis=1)
        q = len(np.unique(outp))
        conf=np.zeros((q,q))
        for p in range(len(outp)):
            conf[outp[p]][pred[p]] += 1
        return conf

class LSTM_lyr: #LSTM lyr Class
    def __init__(self,dim_in,neur_N,beta):
        self.dim_in = dim_in
        self.neur_N = neur_N
        self.beta=beta
        self.Last_Act=None
        self.err_lyr=None
        self.delta_lyr=None
        self.prevU_f, self.prevU_i, self.prevU_c, self.prevU_o= 0,0,0,0

        self.XUD=np.sqrt(6/(dim_in+neur_N))


        self.Wf = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N)) # forget gate
        self.Wi = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N)) # input gate
        self.Wc = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))  # cell gate
        self.Wo = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))  # output gate


```

```
    def act_f(self, x,act):
        if(act == 'softmax'):
            e_x = np.exp(x - np.max(x))
            return e_x/np.sum(e_x, axis=1, keepdims=True)
        elif(act == 'tanh'):
            return np.tanh(x*self.beta)
        elif(act=="sigmoid"):
            return np.exp(2*x)/(1+np.exp(2*x))
        elif(act=="relu"):
            return np.maximum(0,x)
        else:
            return x


    def act_N(self,x, w, act):
        samp_N = len(x)
        inp_temp = np.concatenate((x, -np.ones((samp_N, 1))), axis=1)
        self.Last_Act = self.act_f(inp_temp.dot(w),act)
        return self.Last_Act

    def rec_Act(self,x,hid, act):
        samp_N = len(x)
        inp_temp = np.concatenate((x, -np.ones((samp_N, 1))), axis=1)
        last=hid.dot(self.W2) + inp_temp.dot(self.all_W)
        self.Last_Act = self.act_f(last,act)
        return self.Last_Act

    def act_derv(self, x,act):
        if(act=="sigmoid" or act == 'softmax'):
            return x*(1-x)
        elif (act=="relu"):
            return (x>0)*1
        elif(act == 'tanh'):
            return self.beta*(1-x*x)
        else:
            return np.ones(x.shape)

class LSTM_:
    def __init__(self,data_train):

        self.lyrs=[]

    def lyr_add(self,lyr):
        self.lyrs.append(lyr)

    def Forward(self,data_train):

```

```
        sampN, sampT, sampD = data_train.shape
        sampH=128
        lyr = self.lyrs[0]

        mem = np.empty((sampN, sampT, sampH))
        i_t = mem
        f_t = mem
        c_t = mem
        o_t = mem
        tanhc = mem


        h_t_1=np.zeros((sampN, sampH))
        c_prv=h_t_1


        z = np.empty((sampN, sampT, sampD + sampH))



        #Apply  functions
        for i in range(sampT):
            z[:, i] = np.concatenate((h_t_1, data_train[:, i]),axis=1)
            zt = z[:, i]

            i_t[:, i] = lyr.act_N(zt, lyr.Wi, "sigmoid")
            f_t[:, i] = lyr.act_N(zt , lyr.Wf, "sigmoid")
            c_t[:, i] = lyr.act_N(zt, lyr.Wc, "tanh")
            o_t[:, i] = lyr.act_N(zt, lyr.Wo, "sigmoid")

            mem[:, i] = c_t[:, i]*i_t[:, i] +  c_prv*f_t[:, i]
            c_prv=mem[:, i]

            tanhc[:, i] = lyr.act_f(c_prv, "tanh")
            h_t_1 = o_t[:, i] * tanhc[:, i]


            cac = {"z_summ": z, #Summation of h_t-1 and x_t
                     "memory": mem,  #Memory
                     "tanhc": (tanhc), # tanh memor
                     "f_t": f_t, # f_t out
                     "i_t": (i_t), # i_t out
                     "c_t": (c_t),# c_t out
                     "o_t": (o_t)}# o_t out


        for lyr in self.lyrs[1:len(self.lyrs)]:
```

```
            #For MLP lyrs
            h_t_1=lyr.act_N(h_t_1)
        OUT= h_t_1
        return cac,OUT

    def back(self,l_rate,size_batch,data_train,lab_train,momentum):
        cac,output = self.Forward(data_train)
        out_forw = output
        z = cac["z_summ"]
        c=cac["memory"]
        tanhc=cac["tanhc"]
        f_t=cac["f_t"]
        i_t=cac["i_t"]
        c_t=cac["c_t"]
        o_t=cac["o_t"]

        for q in reversed(range(len(self.lyrs))):# backprop til LSTM part
            lyr = self.lyrs[q]

            if(lyr == self.lyrs[-1]): #output
                lyr.delta_lyr=lab_train-out_forw
            elif(lyr==self.lyrs[0]):
                layer_next = self.lyrs[q+1]
                lyr.err_lyr = layer_next.delta_lyr.dot(layer_next.all_W[0:len(layer_next.all_W)-1].T)
                lyr.delta_lyr=lyr.err_lyr

            else:
                layer_next = self.lyrs[q+1]
                lyr.err_lyr = layer_next.delta_lyr.dot(layer_next.all_W[0:len(layer_next.all_W)-1].T)
                derv=lyr.act_derv(lyr.Last_Act)
                lyr.delta_lyr=derv*lyr.err_lyr


        dWf, dWi, dWc, dWo = 0,0,0,0  #init grads to zero
        H=128
        T = z.shape[1]
        samp_N = len(data_train)

        init_lyr=self.lyrs[0]
        delta=init_lyr.delta_lyr

        for t in reversed(range(T)):#BPTT OF LSTM
            u = z[:, t]

            if t > 0:
```

```python
                c_prv = c[:, t - 1]
            else:
                c_prv = 0


            dc = delta * o_t[:, t] * init_lyr.act_derv(tanhc[:, t],"tanh")

            dc_t = dc * i_t[:, t] * init_lyr.act_derv(c_t[:, t],"tanh")
            di_t = dc * c_t[:, t] * init_lyr.act_derv(i_t[:, t],"sigmoid")
            df_t = dc * c_prv * init_lyr.act_derv(f_t[:, t],"sigmoid")
            do_t = delta * tanhc[:, t] * init_lyr.act_derv(o_t[:, t],"sigmoid")


            dWc += np.concatenate((u, -np.ones((samp_N, 1))), axis=1).T.dot(dc_t)
            dWi += np.concatenate((u, -np.ones((samp_N, 1))), axis=1).T.dot(di_t)
            dWf += np.concatenate((u, -np.ones((samp_N, 1))), axis=1).T.dot(df_t)
            dWo += np.concatenate((u, -np.ones((samp_N, 1))), axis=1).T.dot(do_t)

            # upd gradients
            duc = dc_t.dot(init_lyr.Wc.T[:, :H])
            dui = di_t.dot(init_lyr.Wi.T[:, :H])
            duf = df_t.dot(init_lyr.Wf.T[:, :H])
            duo = do_t.dot(init_lyr.Wo.T[:, :H])

            delta = duc+dui+duf+duo


        for i in range(len(self.lyrs)): #update the Weights
            lyr = self.lyrs[i]
            if(i == 0):

                up_f,up_i,up_c,up_o=np.array([dWf,dWi,dWc,dWo])*l_rate/size_batch

                lyr.Wf+= up_f + lyr.prevU_f*momentum
                lyr.Wi+= up_i + lyr.prevU_i*momentum
                lyr.Wc+= up_c + lyr.prevU_c*momentum
                lyr.Wo+= up_o + lyr.prevU_o*momentum

                lyr.prevU_f,lyr.prevU_i,lyr.prevU_c,lyr.prevU_o=np.array([up_f,up_i,up_c,up_o])

            else:
                samp_N = len(self.lyrs[i - 1].Last_Act)
                inp_temp=np.concatenate((self.lyrs[i - 1].Last_Act, -np.ones((samp_N, 1))), axis=1)
                upd =  (inp_temp.T.dot(lyr.delta_lyr))*l_rate/size_batch
                lyr.all_W+= upd + lyr.prevU*momentum
```

```python
                lyr.prevU = upd

    def train(self,l_rate,size_batch,data_train,lab_train, inp_test, lab_test,
        N_ep,momentum):
        cr_los_list = []
        train_res=[]
        for ep in range(N_ep):
            print("ep:",ep)
            ind=np.random.permutation(len(data_train))

            data_train=data_train[ind]
            lab_train=lab_train[ind]
            batch_N = int(len(data_train)/size_batch)
            for j in range(batch_N):
                train_data = data_train[j*size_batch:size_batch*(j+1)]
                train_labels = lab_train[j*size_batch:size_batch*(j+1)]
                self.back(l_rate,size_batch,train_data,train_labels,momentum)
            _, out_val = self.Forward(inp_test)
            _, out_train = self.Forward(data_train)
            cr_los = np.sum(-np.log(out_val) * lab_test)/len(out_val)
            cr_los1 = np.sum(-np.log(out_train) * lab_train)/len(out_train)
            print('C-E Error of Validation', cr_los)
            print('C-E Error of Train', cr_los1)
            cr_los_list.append(cr_los)
            train_res.append(cr_los1)
        return cr_los_list, train_res

    def Predict(self,inps,realout):
        _,out = self.Forward(inps)
        out = out.argmax(axis=1)
        realout = realout.argmax(axis=1)
        return ((out == realout).mean()*100)


    def conf(self,inp,outp):
        _,pred= self.Forward(inp)
        pred = pred.argmax(axis=1)
        outp = outp.argmax(axis=1)
        q = len(np.unique(outp))
        conf=np.zeros((q,q))
        for p in range(len(outp)):
            conf[outp[p]][pred[p]] += 1
        return conf

class GRU_lyr: #GRU lyr Class
    def __init__(self,dim_in,neur_N,beta):
        self.dim_in = dim_in
```

```
        self.neur_N = neur_N
        self.beta = beta

        self.Last_Act=None
        self.err_lyr=None
        self.delta_lyr=None
        self.prevU_Wz,self.prevU_Wr,self.prevU_Wh=0,0,0
        self.prevU_Uz,self.prevU_Ur,self.prevU_Uh=0,0,0

        self.XUD=np.sqrt(6/(dim_in+neur_N))
        self.w1=np.sqrt(6/(neur_N+neur_N))

        self.Uz = np.random.uniform(-self.w1, self.w1, size=(neur_N, neur_N))
        self.Wz = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))

        self.Ur = np.random.uniform(-self.w1, self.w1, size=(neur_N, neur_N))
        self.Wr = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))

        self.Uh = np.random.uniform(-self.w1, self.w1, size=(neur_N, neur_N))
        self.Wh = np.random.uniform(-self.XUD,self.XUD,(dim_in+1,neur_N))


    def act_f(self, x,act):
        if(act == 'softmax'):
            e_x = np.exp(x - np.max(x))
            return e_x/np.sum(e_x, axis=1, keepdims=True)
        elif(act == 'tanh'):
            return np.tanh(x*self.beta)
        elif(act=="sigmoid"):
            return np.exp(x)/(1+np.exp(x))
        elif(act=="relu"):
            return np.maximum(0,x)
        else:
            return x

    def act_N(self,x, w, h, u, act):
        samp_N = len(x)
        inp_temp = np.concatenate((x, -np.ones((samp_N, 1))), axis=1)
        self.Last_Act = self.act_f(inp_temp.dot(w)+h.dot(u),act)
        return self.Last_Act

    def act_derv(self, x,act):
        if(act=="sigmoid" or act == 'softmax'):
            return x*(1-x)
        elif (act=="relu"):
            return (x>0)*1
        elif(act == 'tanh'):
```

```
            return self.beta*(1-x*x)
        else:
            return np.ones(x.shape)


class GRU_:
    def __init__(self,data_train):


        self.lyrs=[]
    def lyr_add(self,lyr):
        self.lyrs.append(lyr)

    def Forward(self,data_train):

        lyr = self.lyrs[0]     #GRU First lyr
        sampN, sampT, _ = data_train.shape
        sampH=128
        h_t_1=np.zeros((sampN, sampH))
        h_t = np.empty((sampN, sampT, sampH))
        h_t_t = np.empty((sampN, sampT, sampH))

        z_t = np.empty((sampN, sampT, sampH))
        r_t = np.empty((sampN, sampT, sampH))

        #apply funcs
        for t in range(sampT):
            x = data_train[:, t]
            z_t[:, t] = lyr.act_N(x,  lyr.Wz, h_t_1, lyr.Uz , "sigmoid")
            r_t[:, t] = lyr.act_N(x, lyr.Wr, h_t_1,  lyr.Ur , "sigmoid")
            h_t_t[:, t] = lyr.act_N(x, lyr.Wh, (r_t[:, t] * h_t_1), lyr.Uh, "tanh")
            h_t[:, t] = (1 - z_t[:, t]) * h_t_1 + z_t[:, t] * h_t_t[:, t]
            h_t_1 = h_t[:, t]

            cac = {"z_t": z_t,
                     "r_t": r_t,
                     "h_t_t": (h_t_t),
                     "h_t": h_t}


        for ly in self.lyrs[1:len(self.lyrs)]: # MLP
            h_t_1=ly.act_N(h_t_1)

        oup= h_t_1
        return cac,oup
```

```python
    def back(self,l_rate,size_batch,data_train,lab_train,momentum):
        cac,OUT = self.Forward(data_train)
        out_forw = OUT
        z_t = cac["z_t"]
        r_t=cac["r_t"]
        h_t_t=cac["h_t_t"]
        h_t=cac["h_t"]


        for q in reversed(range(len(self.lyrs))):# backprop til GRU
            lyr = self.lyrs[q]
            #outputlyr
            if(lyr == self.lyrs[-1]):
                lyr.delta_lyr=lab_train-out_forw
            elif(lyr==self.lyrs[0]):
                layer_next = self.lyrs[q+1]
                lyr.err_lyr =
                ↪  layer_next.delta_lyr.dot(layer_next.all_W[0:len(layer_next.all_W)-1].T)
                lyr.delta_lyr=lyr.err_lyr

            else:
                layer_next = self.lyrs[q+1]
                lyr.err_lyr =
                ↪  layer_next.delta_lyr.dot(layer_next.all_W[0:len(layer_next.all_W)-1].T)
                derv=lyr.act_derv(lyr.Last_Act)
                lyr.delta_lyr=derv*lyr.err_lyr
        # initialize gradients to zero
        dWz,dWr,dWh = 0,0,0
        dUz,dUr,dUh = 0,0,0


        sampH=128
        samp_N, sampT, sampD = data_train.shape


        init_lyr=self.lyrs[0]
        delta=init_lyr.delta_lyr

        for t in reversed(range(sampT)):
            x = data_train[:, t]
            if t > 0:
                h_t_1 = h_t[:, t - 1]
            else:
                h_t_1 = np.zeros((samp_N, sampH))


```

```
            dz =  (h_t_t[:, t] - h_t_1) * init_lyr.act_derv(z_t[:, t],"sigmoid") *delta
            dh_t_t =  z_t[:, t] * init_lyr.act_derv(h_t_t[:, t],"tanh")*delta
            dr = dh_t_t.dot(init_lyr.Uh.T) * init_lyr.act_derv(r_t[:, t],"sigmoid")* h_t_1

            dWz += np.concatenate((x, -np.ones((samp_N, 1))), axis=1).T.dot(dz)
            dWh += np.concatenate((x, -np.ones((samp_N, 1))), axis=1).T.dot(dh_t_t)
            dWr +=np.concatenate((x, -np.ones((samp_N, 1))), axis=1).T.dot(dr)

            dUz += h_t_1.T.dot(dz)
            dUh += h_t_1.T.dot(dh_t_t)
            dUr += h_t_1.T.dot(dr)


            # update the  gradients


            d9 =  (1 - z_t[:, t])*delta
            d11 = dz.dot(init_lyr.Uz.T)
            d13 = dh_t_t.dot(init_lyr.Uh.T) * (r_t[:, t] + h_t_1 * init_lyr.act_derv(r_t[:, t],"sigmoid").dot(init_lyr.Ur.T))

            delta = d9+d11+d13

        for i in range(len(self.lyrs)):
            lyr = self.lyrs[i]
            if(i == 0):

                up_Wz,up_Wr,up_Wh=np.array([dWz,dWr,dWh])*l_rate/size_batch
                up_Uz,up_Ur,up_Uh=np.array([dUz,dUr,dUh])*l_rate/size_batch

                lyr.Wz+= up_Wz + lyr.prevU_Wz*momentum
                lyr.Uz+= up_Uz + lyr.prevU_Uz*momentum
                lyr.Wr+= up_Wr + lyr.prevU_Wr*momentum
                lyr.Ur+= up_Ur + lyr.prevU_Ur*momentum
                lyr.Wh+= up_Wh + lyr.prevU_Wh*momentum
                lyr.Uh+= up_Uh + lyr.prevU_Uh*momentum


                lyr.prevU_Wz ,lyr.prevU_Wr,lyr.prevU_Wh=np.array([up_Wz,up_Wr,up_Wh])
                lyr.prevU_Uz ,lyr.prevU_Ur,lyr.prevU_Uh=np.array([up_Uz,up_Ur,up_Uh])

            else:
```

```
                samp_N = len(self.lyrs[i - 1].Last_Act)
                inp_temp=np.concatenate((self.lyrs[i - 1].Last_Act,
                 ↪  -np.ones((samp_N, 1))), axis=1)
                upd =  (inp_temp.T.dot(lyr.delta_lyr))*l_rate/size_batch
                lyr.all_W+= upd + lyr.prevU*momentum
                lyr.prevU = upd

    def train(self,l_rate,size_batch,data_train,lab_train, inp_test, lab_test,
     ↪  N_ep,momentum):
        cr_los_list = []
        train_res=[]
        for ep in range(N_ep):
            print("ep:",ep)
            ind=np.random.permutation(len(data_train))

            data_train=data_train[ind]
            lab_train=lab_train[ind]
            batch_N = int(len(data_train)/size_batch)
            for j in range(batch_N):
                train_data = data_train[j*size_batch:size_batch*(j+1)]
                train_labels = lab_train[j*size_batch:size_batch*(j+1)]
                self.back(l_rate,size_batch,train_data,train_labels,momentum)
            _, out_val = self.Forward(inp_test)
            _, out_train = self.Forward(data_train)
            cr_los = np.sum(-np.log(out_val) * lab_test)/len(out_val)
            cr_los1 = np.sum(-np.log(out_train) * lab_train)/len(out_train)
            print('C-E Error of Validation', cr_los)
            print('C-E Error of Train', cr_los1)
            cr_los_list.append(cr_los)
            train_res.append(cr_los1)
        return cr_los_list, train_res

    def Predict(self,inps,realout):
        _,out = self.Forward(inps)
        out = out.argmax(axis=1)
        realout = realout.argmax(axis=1)
        return ((out == realout).mean()*100)


    def conf(self,inp,outp):
        _,pred= self.Forward(inp)
        pred = pred.argmax(axis=1)
        outp = outp.argmax(axis=1)
        q = len(np.unique(outp))
        conf=np.zeros((q,q))
        for p in range(len(outp)):
            conf[outp[p]][pred[p]] += 1
```

```
        return conf



def q1():
    filename = 'data1.h5'
    data=h5py.File(filename, 'r')["data"][()]
    data_gray = 0.2126*data[:,0,:,:] + 0.7152*data[:,1,:,:] +
    ↪  0.0722*data[:,2,:,:]
    mean_data = np.mean(data_gray, axis=(1,2))

    for k in range(len(mean_data)):
        data_gray[k,:,:] -= mean_data[k]

    std_data = np.std(data_gray)

    normalized = np.clip(data_gray, std_data*(-3),std_data*3)

    normalization = lambda x:(x-x.min()) / (x.max()-x.min())

    data_nor= normalization(normalized)*0.8 + 0.1

    data_T=np.transpose(data,(0,2,3,1))
    random_sample=np.random.randint(0,len(data_nor),size=(200))

    row = 10
    col = 20
    fig=plt.figure(figsize=(20, 10), dpi= 100)
    for j in range(row*col):
        plt.subplot(row,col,j+1)
        rand_pic=random_sample[j]
        plt.imshow(data_T[rand_pic,:,:,:])
        plt.axis('off')
    plt.show()

    fig=plt.figure(figsize=(20, 10), dpi= 100)
    for j in range(row*col):
        plt.subplot(row,col,j+1)
        rand_pic=random_sample[j]
        plt.imshow(data_nor[rand_pic,:,:],cmap='gray')
        plt.axis("off")
    plt.show()


    num_pixel=data_nor.shape[1]
    Lin = Lout = num_pixel**2
    Lhid = 64
```

```
    batch_size = 32
    epoch = 200
    lmb = 5e-4
    alpha = 0.85
    rho = 0.025
    eta = 0.075
    beta = 2

    params= (Lin, Lhid, lmb, beta, rho)

    data_solve=data_nor.reshape(data_nor.shape[0],data_nor.shape[1]**2)
    w,j=solver(data_solve, params, eta, alpha, epoch, batch_size)

    plot_weights(w)
    hid_list=[16,49,100]
    lmb_list=[0,2e-4,1e-3]
    j_list=[j]
    w_list=[w]
    for i in range(len(hid_list)):
        for j in range(len(lmb_list)):
            params= (Lin, hid_list[i], lmb_list[j], beta, rho)
            print("parameters: Lin={}, Lhid={}, lmb={}, beta={}, rho={}
             ↪  ".format(Lin, hid_list[i], lmb_list[j], beta, rho))
            w,j=solver(data_solve, params, eta, alpha, epoch, batch_size)
            j_list.append(j)
            w_list.append(w)
    for i in w_list[1:]:
        plot_weights(i)


def q2():
    filename="data2.h5"
    testx = h5py.File(filename, 'r')["testx"][()]
    traind = h5py.File(filename, 'r')["traind"][()]
    trainx = h5py.File(filename, 'r')["trainx"][()]
    vald = h5py.File(filename, 'r')["vald"][()]
    valx = h5py.File(filename, 'r')["valx"][()]
    words = h5py.File(filename, 'r')["words"][()]
    v_size = 250
    lr = 0.15
    moment = 0.85
    batch = 200
    epoch = 50



    valx1h = data_1h(valx, v_size)
```

```
vald1h = label_1h(vald, v_size)

P_1 = 256
D_1 = 32

model1 = Neural()
model1.layer_add(Layer(D_1, v_size,0,0.25, 'emb'))
model1.layer_add(Layer(3*D_1, P_1, 0,0.25, 'sigmoid'))
model1.layer_add(Layer(P_1, v_size, 0,0.25,'softmax'))

loss1 = model1.Train(lr,batch,trainx,traind,valx1h,vald1h,epoch,moment,v_size)

P_2 = 128
D_2 = 16

model2 = Neural()
model2.layer_add(Layer(D_2, v_size,0,0.25, 'emb'))
model2.layer_add(Layer(3*D_2, P_2, 0,0.25, 'sigmoid'))
model2.layer_add(Layer(P_2, v_size, 0,0.25,'softmax'))

loss2 = model2.Train(lr,batch,trainx,traind,valx1h,vald1h,epoch,moment,v_size)


P_3 = 64
D_3 = 8

model3 = Neural()
model3.layer_add(Layer(D_3, v_size,0,0.25, 'emb'))
model3.layer_add(Layer(3*D_3, P_3, 0,0.25, 'sigmoid'))
model3.layer_add(Layer(P_3, v_size, 0,0.25,'softmax'))

loss3 = model3.Train(lr,batch,trainx,traind,valx1h,vald1h,epoch,moment,v_size)

plt.plot(loss1,label="D=32,P=256")
plt.plot(loss2,label="D=16,P=128")
plt.plot(loss3,label="D=8,P=64")
plt.legend()
plt.title('Cross Entropy Error Plots For Different (D,P) Values')
plt.xlabel('Number of Epochs')
plt.ylabel('Cross Entropy Error')
plt.show()


rnd = np.random.permutation(len(testx))[0:5]
```

```
    samp = testx[rnd,:]


    samp1h = data_1h(samp, 250)

    preds1 = model1.predict(samp1h, 10)

    for i in range(5):
        print('Triagram {}: \n'.format(i+1))
        st=""
        for q in range(3):
            st+=str(words[samp[i,q]-1].decode("utf-8"))+" "

        print(st+"\n")
        print('Most Probable 10 Predictions are:\n ')
        w_list=[]
        for j in range(10):
            w_list.append(str(words[preds1[j,i]-1].decode("utf-8")))
        for j in w_list:
            print(j)
        print("\n")


    preds2 = model2.predict(samp1h, 10)

    for i in range(5):
        print('Triagram {}: \n'.format(i+1))
        st=""
        for q in range(3):
            st+=str(words[samp[i,q]-1].decode("utf-8"))+" "

        print(st+"\n")
        print('Most Probable 10 Predictions are:\n ')
        w_list=[]
        for j in range(10):
            w_list.append(str(words[preds2[j,i]-1].decode("utf-8")))
        for j in w_list:
            print(j)
        print("\n")


    preds3 = model3.predict(samp1h, 10)

    for i in range(5):
        print('Triagram {}: \n'.format(i+1))
        st=""
```

```
        for q in range(3):
            st+=str(words[samp[i,q]-1].decode("utf-8"))+" "

        print(st+"\n")
        print('Most Probable 10 Predictions are:\n ')
        w_list=[]
        for j in range(10):
            w_list.append(str(words[preds3[j,i]-1].decode("utf-8")))
        for j in w_list:
            print(j)
        print("\n")
def q3(t):
    file = h5py.File("data3.h5", "r") # get data

    dat_tra = np.array(file[list(file.keys())[0]])
    lab_tra = np.array(file[list(file.keys())[1]])
    dat_test = np.array(file[list(file.keys())[2]])
    lab_test = np.array(file[list(file.keys())[3]])



    neuron_n=128
    batch = 32
    epoch = 30
    alpha = 0.85 # momentum
    eta = 0.03 #learning rate
    ind=np.random.permutation(len(dat_tra))
    dat_tra=dat_tra[ind]
    lab_tra=lab_tra[ind]

    size_v = int(len(dat_tra) / 10)
    dat_v=dat_tra[:size_v]
    lab_v=lab_tra[:size_v]
    dat_tra1=dat_tra[size_v:]
    lab_tra1=lab_tra[size_v:]

    if t==1:
        model_rnn= RNN_(dat_tra1)
        model_rnn.lyr_add(lyr(3, neuron_n,'hyperbolic',1)) #3 val sensor
        model_rnn.lyr_add(lyr(neuron_n,70,'relu',1))
        model_rnn.lyr_add(lyr(70,30,'relu',1))
        model_rnn.lyr_add(lyr(30,6,'softmax',1))
        creL_rnn, lis_tra_rnn =
         ↪ model_rnn.train(eta,batch,dat_tra1,lab_tra1,dat_v,lab_v,epoch,alpha)

        plt.plot(creL_rnn)

```

```
        plt.xlabel('Number of Epochs')
        plt.ylabel('C-E Error ')
        plt.title('C-E Error of of Validation for RNN')
        plt.show()

        acc_test_rnn=model_rnn.Predict(dat_test,lab_test)
        acc_train_rnn=model_rnn.Predict(dat_tra1,lab_tra1)
        print("Test set Accuracy: "+str(acc_test_rnn)+"%")
        print("Train set Accuracy: "+str(acc_train_rnn)+"%")




        conf_test_rnn=model_rnn.conf(dat_test,lab_test)
        sn.heatmap(conf_test_rnn, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of test set for rnn")
        plt.ylabel("Real")
        plt.show()

        conf_train_rnn=model_rnn.ConfusionMatrix(dat_tra1,lab_tra1)

        sn.heatmap(conf_train_rnn, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of train set for rnn")
        plt.ylabel("Real")
        plt.show()

    elif t==2:
        model_lstm= LSTM_(dat_tra1)
        model_lstm.lyr_add(LSTM_lyr(131, neuron_n,1)) #3 val from sensor 128 prev
        model_lstm.lyr_add(lyr(neuron_n,70,'relu',1))
        model_lstm.lyr_add(lyr(70,30,'relu',1))
        model_lstm.lyr_add(lyr(30,6,'softmax',1))
        creL_lstm, lis_tra_lstm = model_lstm.train(eta,batch,dat_tra1,lab_tra1,dat_v,lab_v,epoch,alpha)

        plt.plot(creL_lstm)

        plt.xlabel('Number of Epochs')
        plt.ylabel('C-E Error ')
        plt.title('C-E Error ofValidation for LSTM')
        plt.show()

        acc_test_lstm=model_lstm.Predict(dat_test,lab_test)
```

```
        acc_train_lstm=model_lstm.Predict(dat_tra1,lab_tra1)
        print("Test set Accuracy: "+str(acc_test_lstm)+"%")
        print("Train set Accuracy: "+str(acc_train_lstm)+"%")




        conf_test_lstm=model_lstm.conf(dat_test,lab_test)
        sn.heatmap(conf_test_lstm, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of test set for lstm")
        plt.ylabel("Real")
        plt.show()

        conf_train_lstm=model_lstm.ConfusionMatrix(dat_tra1,lab_tra1)
        sn.heatmap(conf_train_lstm, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of train set for lstm")
        plt.ylabel("Real")
        plt.show()
    elif t==3:
        model_gru = GRU_(dat_tra1)
        model_gru.lyr_add(GRU_lyr(3, neuron_n,1))
        model_gru.lyr_add(lyr(neuron_n,70,'relu',1))
        model_gru.lyr_add(lyr(70,30,'relu',1))
        model_gru.lyr_add(lyr(30,6,'softmax',1))
        creL_gru, lis_tra_gru = model_gru.train(eta,batch,dat_tra1,lab_tra1,dat_v,lab_v,epoch,alpha)

        plt.plot(creL_gru)

        plt.xlabel('Number of Epochs')
        plt.ylabel('C-E Error ')
        plt.title('C-E Error ofValidation for GRU')
        plt.show()

        acc_test_gru=model_gru.Predict(dat_test,lab_test)
        acc_train_gru=model_gru.Predict(dat_tra1,lab_tra1)
        print("Test set Accuracy: "+str(acc_test_gru)+"%")
        print("Train set Accuracy: "+str(acc_train_gru)+"%")




        conf_test_gru=model_gru.conf(dat_test,lab_test)
```

```
        sn.heatmap(conf_test_gru, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of test set for gru")
        plt.ylabel("Real")
        plt.show()

        conf_train_gru=model_gru.ConfusionMatrix(dat_tra1,lab_tra1)

        sn.heatmap(conf_train_gru, yticklabels=[1, 2, 3, 4, 5, 6], xticklabels=[1, 2, 3, 4, 5, 6],  cmap=sn.cm.rocket_r, fmt='g')
        plt.xlabel("Prediction")
        plt.title("Confusion Matrix of train set for gru")
        plt.ylabel("Real")
        plt.show()

## in order to run the first question please use q1() function
## in order to run the second question please use q2() function
##in order to run the third question please use the q3(t) function, such that t can only take 1,2 or 3
## those t values represent the part of the third question
## Thus, in order to run RNN code use q3(1)
## in order to run LSTM code use q3(2)
## in order to run GRU code use q3(3)
```