EEE 443- Neural Networks Interim Project Report

Question 1: Autoencoder

a) Gray Scaling

In part 1a, it is asked to illustrate random 200 images from 'data1.h5' which is provided by the manual. This dataset contains 16x16 RGB images is then passed through some processes as instructed. The RBG images are stored as array with shape (sample size, 3, 16,16) where 3 represents the RGB channels. The images are converted to gray scale with luminosity model Y = 02126 x R + 0.7152xG + 0.0722xB for each pixel. Then, all the images are clipped with $\pm$3 std. To prevent saturation, clipped image values are mapped to [0.1, 0.9] and normalized in this range as it is instructed. For first part of this question randomly chosen 200 samples are illustrated in figure 1 and in figure to corresponding gray scaled version.

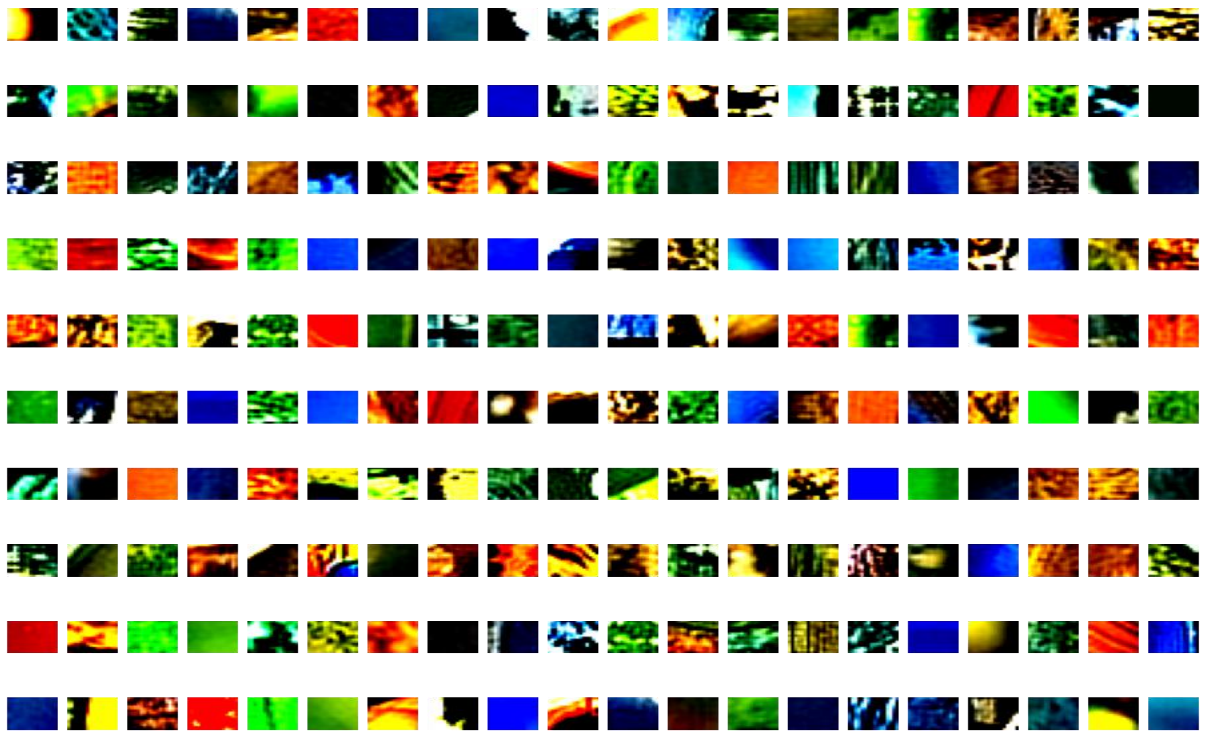

Figure 1: Randomly Chosen 200 RGB Image from 'data1.h5' dataset

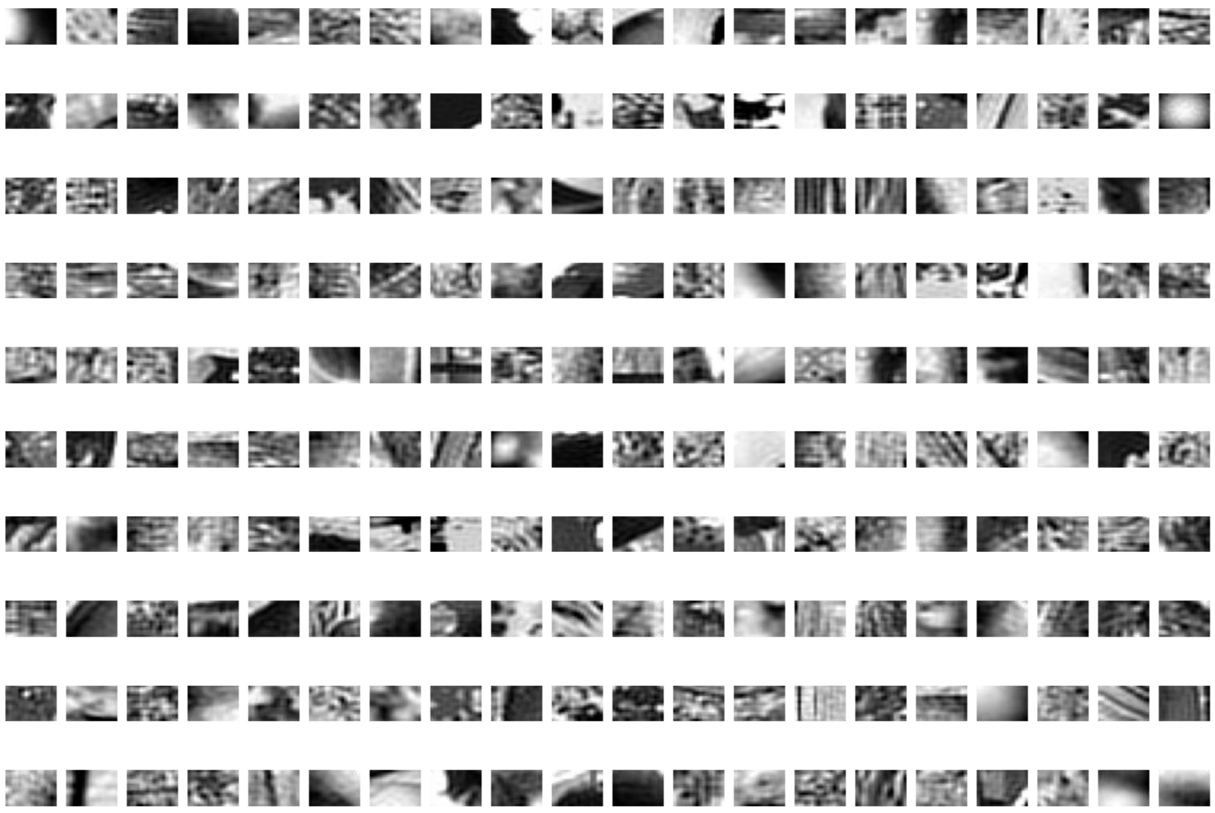

Figure 2: Randomly Chosen 200 RGB Image's Gray Scaled Versions from 'data1.h5' dataset

b) Auto Encoder Implementation

In the second part of question 2, Auto encoder is implemented using class of Python according to the specified parameters instructed in lab manual. In figure 2, autoencoder structure is illustrated for single hidden layer structure similar to question 1 autoencoder.

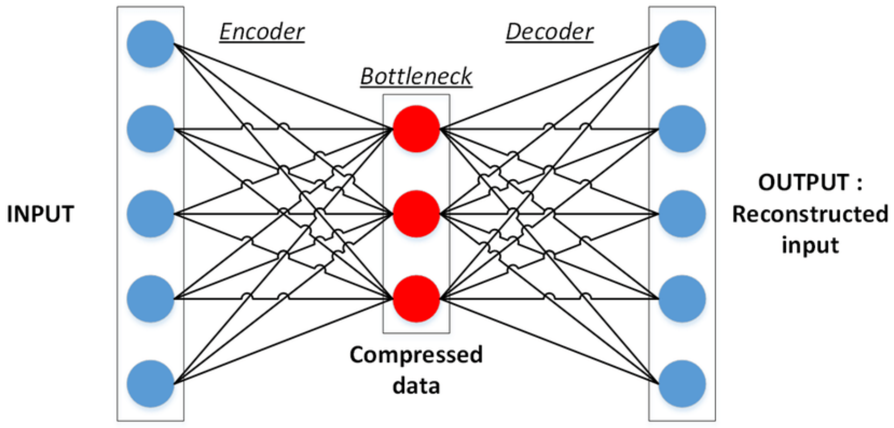

Figure 3: Autoencoder Structure [1]

The weights of the Autoencoder are initialized in interval [-w_o, w_o] where $w_o = \sqrt{6/(L_{pre} + L_{post})}$ where L_i is the connection size (with I denotes the connected part). Due to autoencoder structure weight matrix connected to input equals to transpose of weight matrix connected to output layer. Then, cost function which calculates cost (J), and their derivatives (J_{grad}) was implemented with name of aeCost with inputs; weights, data (with size of L_{in}xN), and parameters (L_{in}, L_{hidden}, λ, β, ρ). The corresponding cost function is given as summation of three terms.

The MSE component, which is the average of l2-norm of the difference between network output and the desired response.

$$\frac{1}{2N}\sum_{i=1}^{N}\|d(m)-o(m)\|^2$$

The other one is Tykhonov regularization on connection weights (λ), and β for the controlling relative weighting;

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

Last term in the summation is Kullback-Leibler (KL) divergence for tuning sparsity level with ρ;

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

$\widehat{\rho_b}$ is the activation average of hidden unit b. Activation function used in algorithm is sigmoid;

$$S(x)=\frac{1}{1+e^{-x}}\ and\ S'(x)=\frac{e^{-x}}{(1+e^{-x})^2}$$

Gradient of losses calculated for three terms and returned in the function and used as input in gradient descent solver for purpose of minimizing this cost. It is also instructed that L_{hid} as 64 and λ as 5x10^{-4} and β, ρ parameters that optimizes loss. With experiments they are chosen as 2.12 and 0.15 correspondingly.

c) First Layer Connection Weights in Hidden Layer

In solver back propagation is applied with chain rule. J_{grad} is used to update weight and bias parameters where $a_i = a_i - learning\ rate \times da_i$ (a as weight or bias matrix). After completing autoencoder, it is trained with determined parameters (batch size=32, L_{hidden}=64, λ=5e-4, β=2.2, ρ=0.15 and learning rate of 0.07). Corresponding weight matrix are illustrated in figure 4.

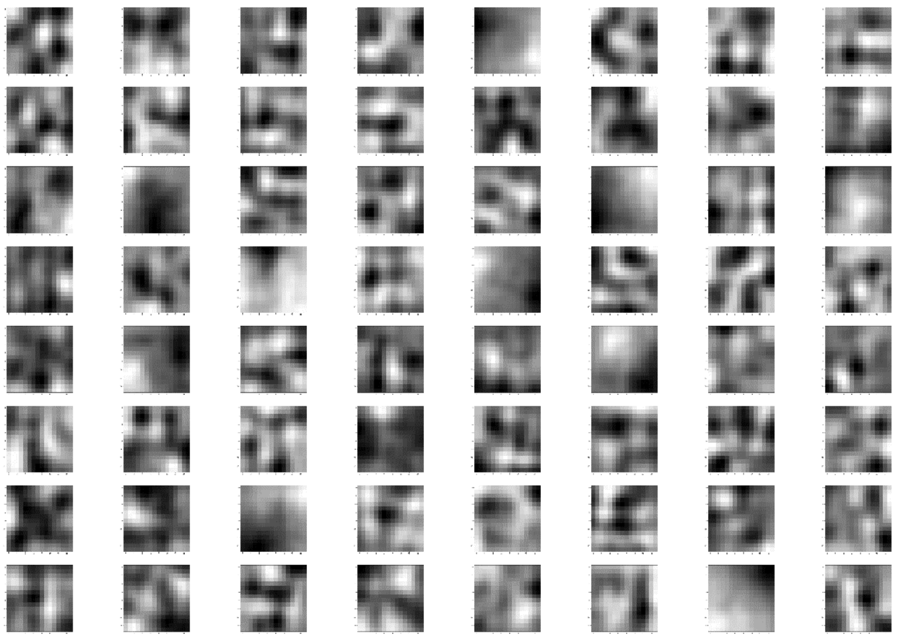

Figure 4: First Layer Connection Weights (λ = 5e-4, L_{hid}= 64)

From figure 4, features of input images lead to some patterns in weights as black and white. Some weights represent similar feature for different images because the autoencoder updates weight to catch repeated features in images to encode existed pattern. Hence, when the encoded pattern is passed through decoder, constructed patterns of encoder are realized by decoder.

d) Training Autoencoder for Three Hidden Layer Size and λ (fixed β and ρ)

For this part chosen L_{hid} values are 36, 49, 81 and λ values are 0, 4e-4, 15e-4. As a result, nine different training was completed.

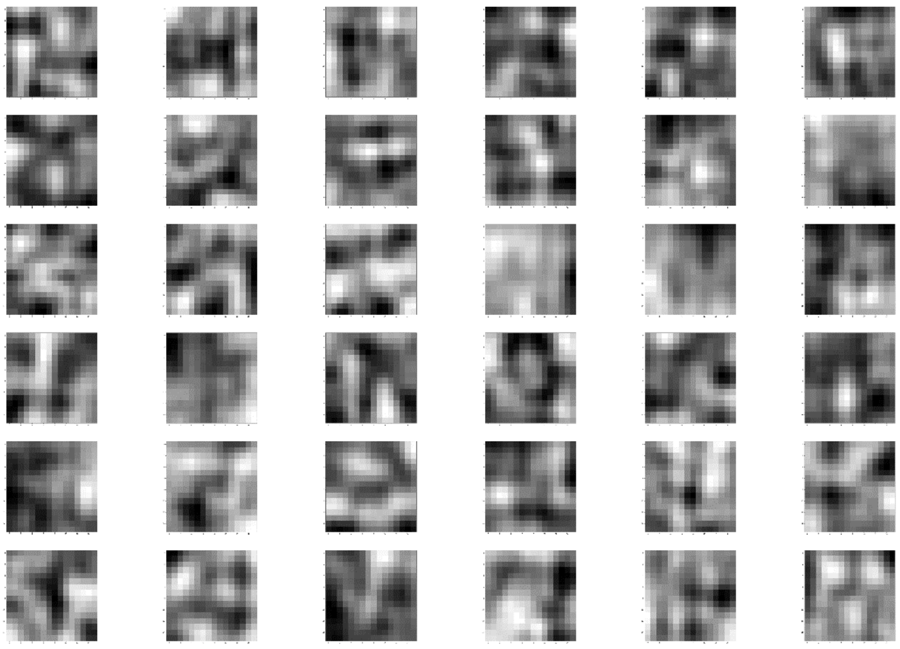

Figure 5: First Layer Connection Weights (λ = 0, L_{hid}= 36)

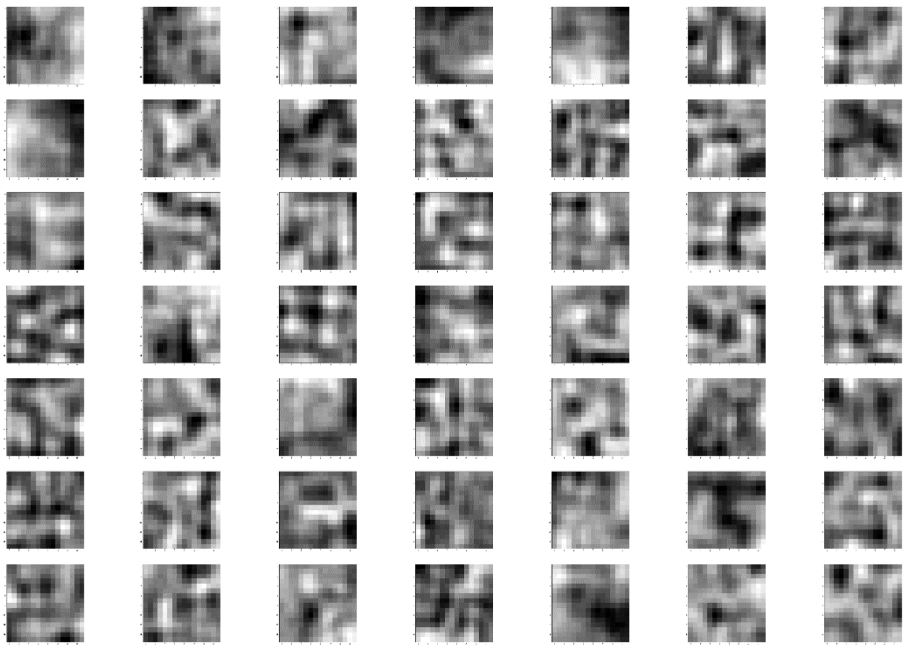

Figure 6: First Layer Connection Weights (λ = 0, L_{hid}= 49)

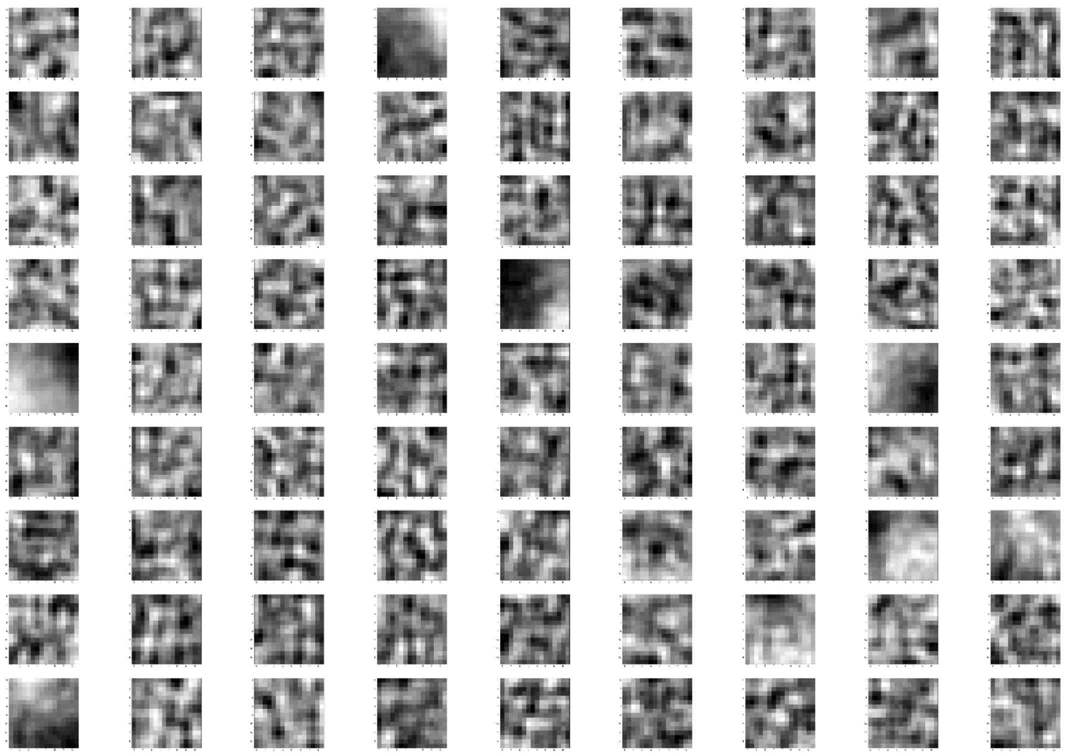

Figure 7: First Layer Connection Weights (λ = 0, L_{hid}= 81)

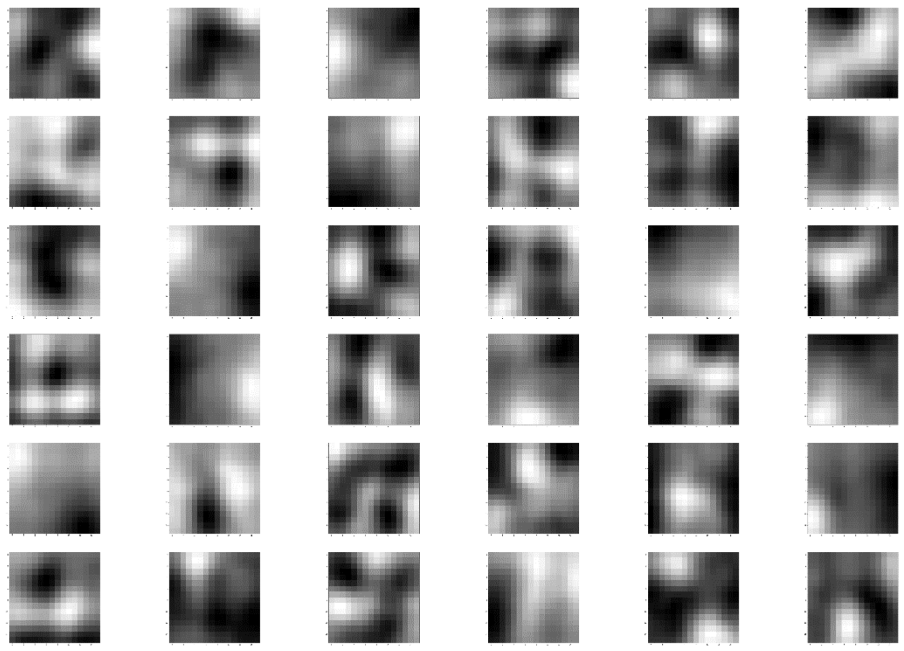

Figure 8: First Layer Connection Weights (λ = 0.001, L_{hid}= 36)

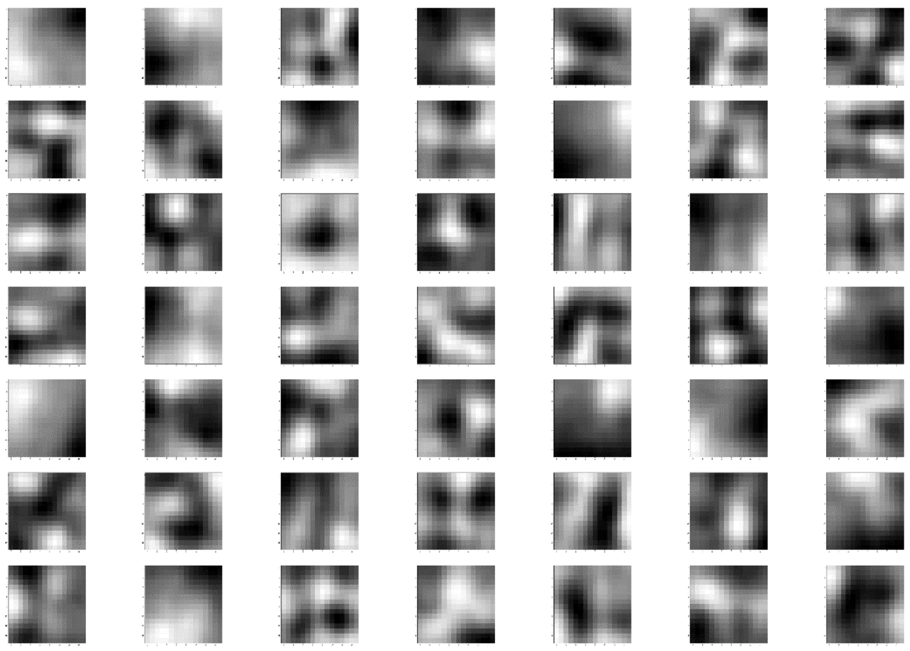

Figure 9: First Layer Connection Weights (λ = 0.001, L_{hid}= 49)

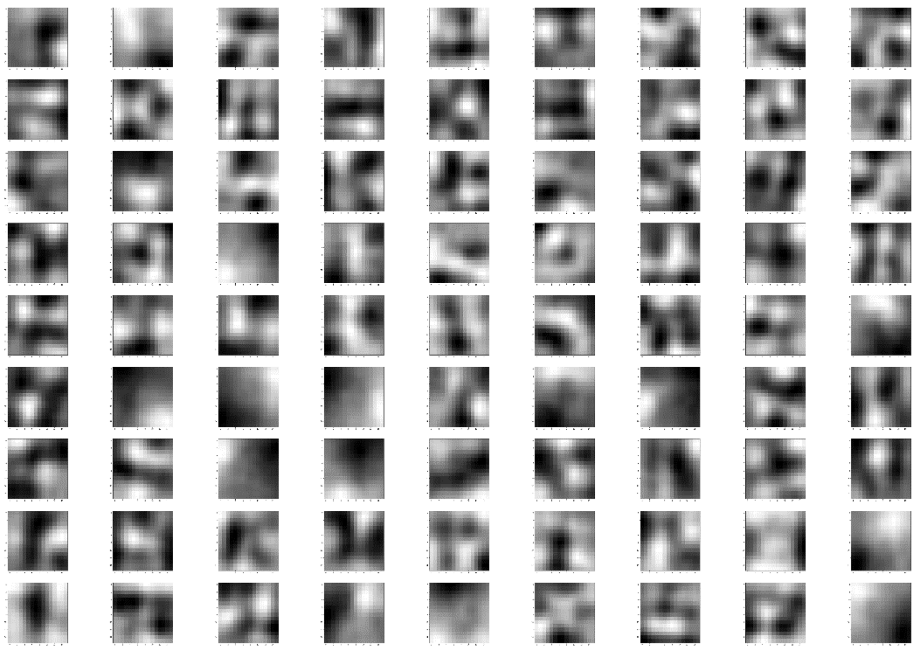

Figure 10: First Layer Connection Weights (λ = 0.001, L_{hid}= 81)

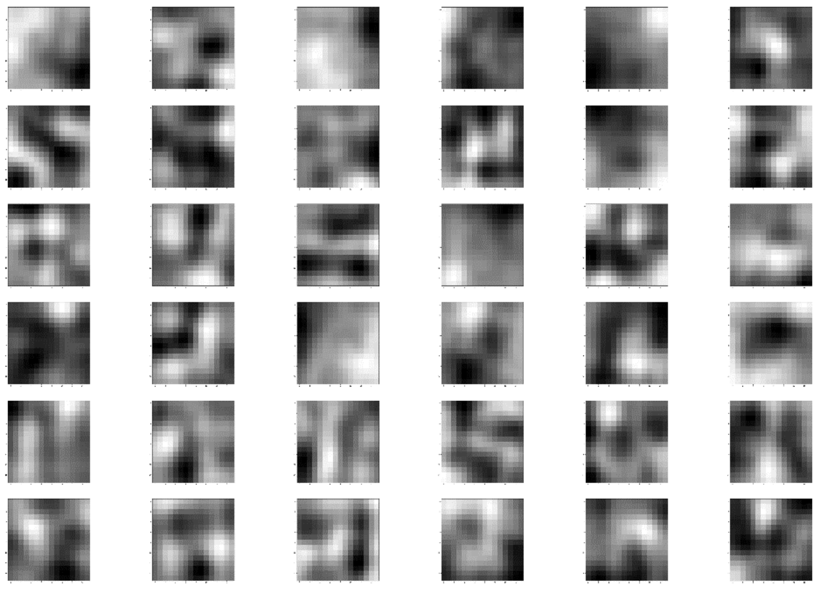

Figure 11: First Layer Connection Weights (λ = 0.0085, L_{hid}= 36)

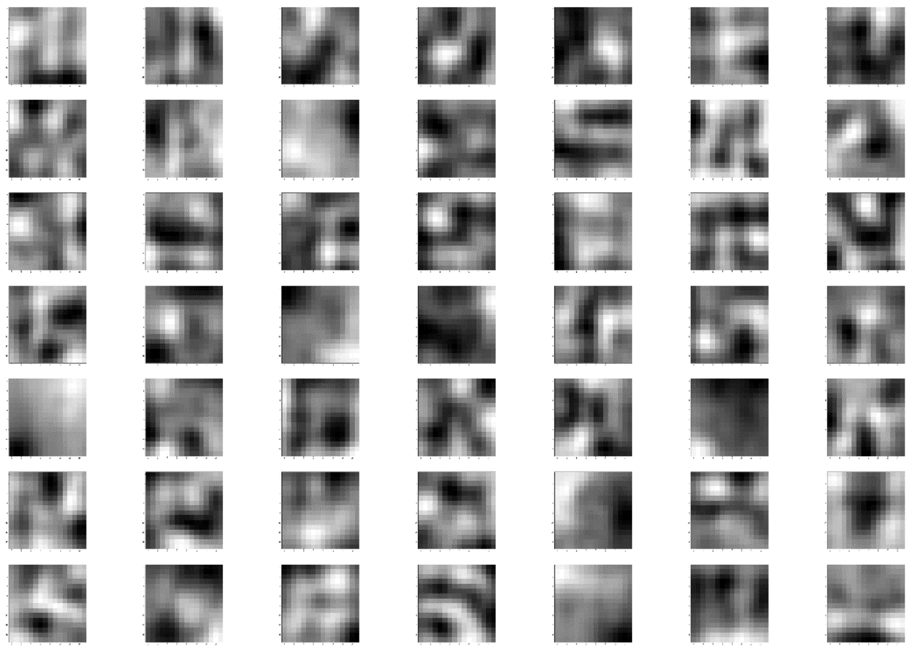

Figure 12: First Layer Connection Weights (λ = 0.0085, L_{hid}= 49)

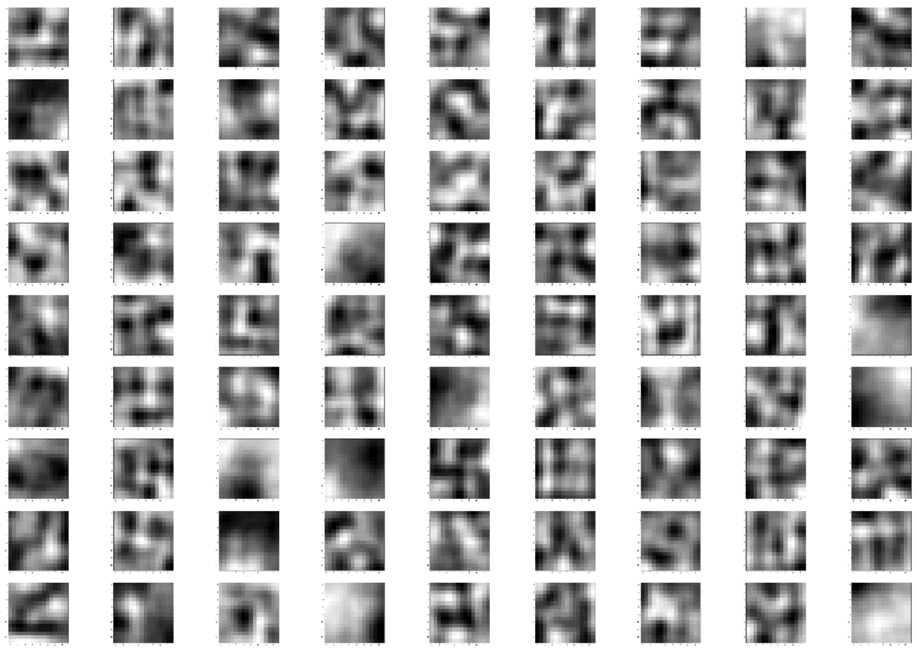

Figure 13: First Layer Connection Weights (λ = 0.0085, L_{hid}= 81)

From figures 5-13, λ is the regularization constant which helps preventing overfitting by reducing noise (make weights sparse). Hence, it affects the noise in hidden unit weights. λ has the best performance among others which shows it prevented the overfitting. Even though, λ is increased higher from 0.001 to 0.0085, they started with higher loss and could not decrease it as much as lower λ value.

As hidden unit size L_{hid} is increased, more features expected to be extracted since the storage is increasing. As a result, different patterns in image created. The effect of hidden unit size can be observed in loss graph which shows overlapping results. However, this increase always not shows better losses. When the optimum value is exceeded, unnecessary features might be realized and create noise by the encoder. With λ=0.0085, the best performance is attained with hidden size 49 from the figure 14 due to redundant future extraction of 81. With higher hidden layer sizes, the algorithms are able decrease loss and fits quicker which results in overfitting and overlook some features of image as in λ=0 case.

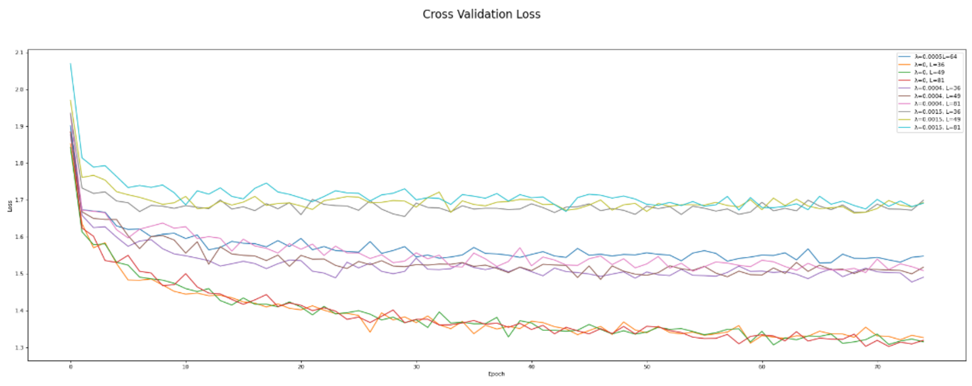

Figure 14: Loss Curves for Different λ and L_{hid} values

The choice of λ and hidden size must be chosen so carefully like extreme values must be avoided so that over and underfitting is prevented. Higher hidden layer sizes led to overfitting, so the optimum value is determined as the middle one. From the loss curve, the best option is chosen as λ=0 and L_{hid}=49 case. Even though, higher λ value expected to perform better, with λ=0.001 the optimum loss is reached at that value.

Question 2: Natural Language Processing Neural Network

a) Neural Network Structure

In the second question, it is asked to implement neural network architecture for natural language processing (NLP). Sequences are words are given in the dataset 'data2.h5' provided by the lab. All the sequences consist of 4 words or punctuations. 250 different vocabulary items are existed in the dataset. With the neural network, it is aimed to guess the 4th word of a sequence. In the dataset, training, validation, and test labels and data is separated by default.

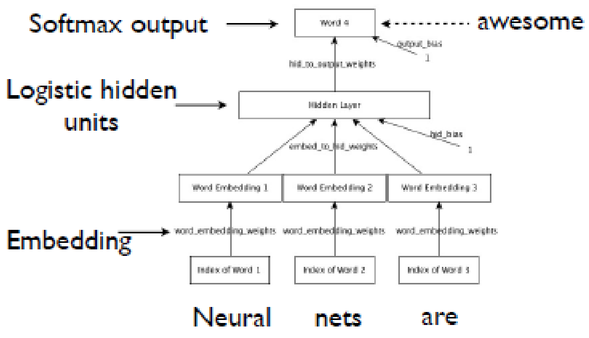

Figure 15: Neural Network Structure

With given structure in figure 15, with using backpropagation the network called as nlp is trained. Three input neurons are used for each word of the input sequence in total 3. The words converted to word embeddings with matrix of R (with size of 250xD) where D varies in question and 250 is the number of dictionary elements. For the creation of word embeddings one hot coded matrix representing words passed through neural network. Hence, for each word in trigram, length of 250 vector is used as an input. Then, the word embeddings passed to hidden layer shown in figure 15. Thee corresponding hidden layer uses sigmoidal activation function ($S(x) = \frac{1}{1+e^{-x}}$). To sum up, first hidden layer used 3x250 vector to represent each word in length D, then passed to 2nd hidden layer with size P to train NLP algorithm with sigmoidal activation function. Hence, the embedding matrix is same for other part. To predict 4th word soft-max activation function is used as ($o_i = \frac{e^{z_i}}{\sum e^{z_i}}$). Cros entropy error is used to stop training when threshold is reached in the validation set ($E_{CE} = -\sum_{j=1}^{M} y_j \times \log{(pred_j)}$).

From the above explanation, the layer size of the network is (750,D,P,250) where 750 is input vector for three words and 250 is the output vector for single word as one hot encoded. So W_0 consists of three of size of (250,D) ((750,D) in total) and its gradient also determined in three separate matrixes. The weight gradients are added and mean founded by division of three. Then, result mean gradient matrix used with momentum multiplier. W_1 has (D,P) and $W_2$is size of (P, 250) and all sampled from normal distribution with zero mean 0.01 variance. In other weights gradients are founded as in tutorials. Training is done with learning rate of 0.15, momentum of 0.85, 50 epoch and mini-batch size of 200 samples with three different (D,P) values as they are (32,256), (16,128), (8,64).

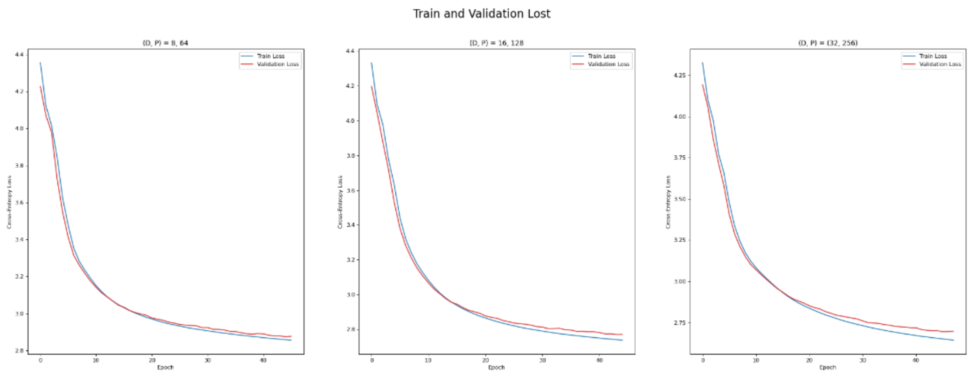

Figure 16: Train and Validation Loss

From the figure 16 and terminal values, it is seen that the networks performed better with higher hidden layer size (with (8,64) loss is 2.9, with (16,126) loss is 2.8 and (32,256) loss is 2.7). First two trials reach convergence around 45 epochs and the final one does not reach to convergence at 50 epochs. Hence, best loss is observed with (32,256) case because it's natural that increased layer size increases performance.

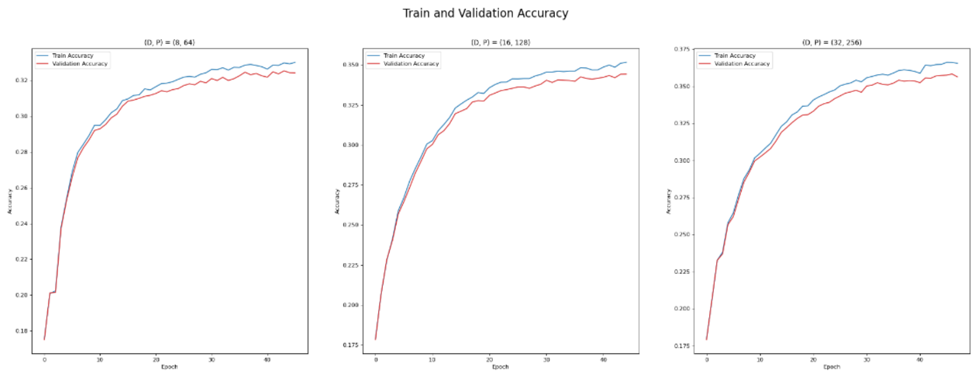

Figure 17: Train and Validation Accuracy

Similar to loss curves in figure 16, the accuracy of training and validation shows that with increasing hidden layer size the accuracy of the system is also increasing with 0.37 best in (32,256). Worst case is with (8,64) case with accuracy 0.31. These accuracies might seem low but since there is 250 options for the network as output, it is as expected. The accuracy is calculated according to most probable option, but as shown in second part some, predicted words are listed with best 10 probabilities. This prediction may be replaceable with each other in logical manner (as an example 'you' and 'they').

b) Testing Network

In this part the most accurate system (32,256) is tested, and the guess of the network seems logical even with 30% accuracy. Ten words with highest probabilities are listed for test trigrams. The test accuracy is found as 31.1%. The possibility of proper other words also decreases logical sequencies in the training. For example, since dataset is not containing 'I like cats more', it avoids predicting 'more' after trigram of 'I like cats' but in real life it makes sense. Even though it predicts 'more' after the trigram, the network is penalized because of the dataset. In similar manner, since every sentence contains a lot of punctuations it pushes towards guessing punctuations, tend to choose repeated words. The network, however, founds interchangeable words and synonyms.

Sequence 1: [b'even'] [b'come'] [b'to']

Label: [[b'the']]

1. [b'me']
2. [b'him']
3. [b'the']
4. [b'them']
5. [b'us']
6. [b'.']
7. [b'it']
8. [b'do']
9. [b'that']
10. [b'this']

Sequence 2: [b'you'] [b'take'] [b'it']

Label: [[b'from']]

1. [b'?']
2. [b'.']
3. [b',']
4. [b'out']
5. [b'to']
6. [b'up']
7. [b'from']
8. [b'all']
9. [b'back']
10. [b'time']

Sequence 3: [b'will'] [b'be'] [b'found']

Label: [[b'.']]

1. [b'.']
2. [b'?']
3. [b',']
4. [b'up']
5. [b'in']
6. [b'the']
7. [b'for']
8. [b'me']
9. [b'them']
10. [b'him']

Sequence 4: [b'know'] [b'who'] [b'they']

Label: [[b'are']]

1. [b'are']
2. [b'were']
3. [b'want']
4. [b'have']
5. [b'can']
6. [b'do']
7. [b'did']
8. [b'know']
9. [b'should']
10. [b'will']

Sequence 5: [b','] [b'what'] [b'do']

Label: [[b'you']]

1. [b'you']
2. [b'we']
3. [b'they']
4. [b'i']
5. [b'nt']
6. [b'it']
7. [b'he']
8. [b'that']
9. [b'she']
10. [b'the']

The below explanation can be explained better with the sequences. As the sequence 3, 4 and 5 are predicted correctly, the sequences 1 and 2 are guessed wrong. In sequence 1, the correct result is predicted in the 3rd place and other all guesses also seem logical. In the 2nd sequence, however the correct label predicted in 7th place and strongest guesses are punctuations which is also kind of logical prediction. In addition, in the correct decision, we can see that other possibilities are close words. As an example, in sequence for correct answer and highest order prediction is ‘are’ followed by ‘were’, ‘have’, ‘can’ etc., similar to sequence 5. Therefore, as a result since most of the predictions make sense, the accuracy can be considered higher than actual. Sometimes meaningless predictions also done as in example 5, since it predicts in 5th place ‘nt’ after ‘, what do’. So, maybe performance metric should be changed to see actual accuracy since network makes meaningful predictions. Also, the decision according to only 3-word input restricts the performance due to insufficient context, so that even sometimes I cannot give meaning to input data. As final result, with bigger hidden layer size performance is increased.

Question 3 Action Classifier Neural Network

In the last question, it is asked to implement network for classification of human activity with three sensors and time sequence consists of 150-time steps. Using back propagation through time algorithm single layer recurrent neural network (RNN), long-short term memory (LSTM) and gated recurrent units (GRU) are implemented which are followed by multi-layer perceptron. For this purpose, a class called sRNN is created. With this class, the weights and biases are created, and training and test is implemented according to model choice of the user. The training is completed when the $\pm$0.015 of mean is reached after a valid number of epochs. For comparison purposes parameter are chosen as mini-batch size of 32, learning rate of 0.1, momentum rate of 0.85 with 50 epochs since it is instructed by the manual and hidden layer size as (16,32),(64,16), and (64,32) for comparison purposes. Also, it is instructed to use Xavier distribution for initializing the weight and bias matrices as W in range [-a,a], b=0. Where r is $\sqrt{6/L_{pre}L_{next}}$. Then, data passed through multi-layer perceptron with softmax function paired with cross-entropy function, and it is asked to use stochastic gradient descent. Hidden multi-layer perceptron uses activation function called ReLU. For each model backpropagation thorough time is used as it is asked.

The back propagation of multi-layer perceptron is as $\frac{dE}{dW} = h^T.\delta$ and $\frac{dE}{db} = [1]_{1xL}.\delta$ where δ is error gradient, E cost, b is bias, and w is weight. Also, weights and biases are updated according to $W_{change} = \sum_{t=t}^{T} \frac{dE(t)}{dW}$. Then, it is multiplied and subtracted from weight matrices with momentum multiplier.

a) Recurrent Neural Network (RNN)

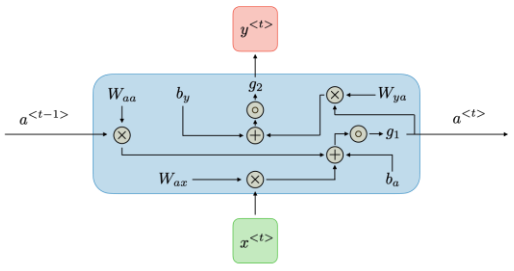

Figure 18: RNN Architecture [2]

Recurrent neural network (RNN) with 128 neurons with hyperbolic tangent activation function is implemented. RNN is different from multi-layer perceptron with its feedback connection, so that it takes the previous output as an input. From that its hidden activation is calculated with $h_t = \tanh(W_{ih}x_t + W_{hh}h_{t-1})$ which is very beneficial for back propagation through time (BPTT). In this task, gradient descent changes as error's partial derivative according to weights are added up through time.

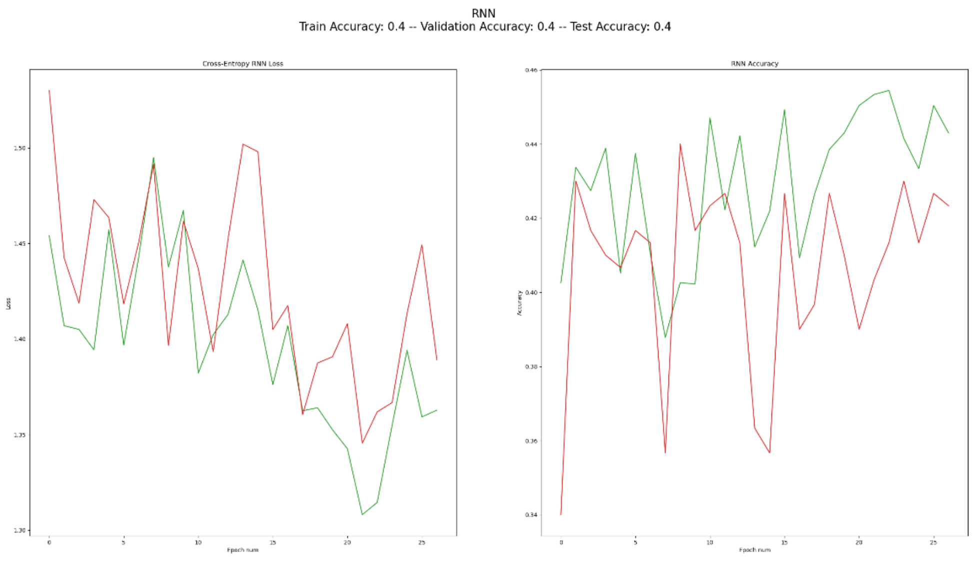

Figure 19: Recurrent Neural Network Losses with Accuracy Test Accuracy: 40.1% (η=0.1,α=0.85,50 epoch, mini batch size of 32)

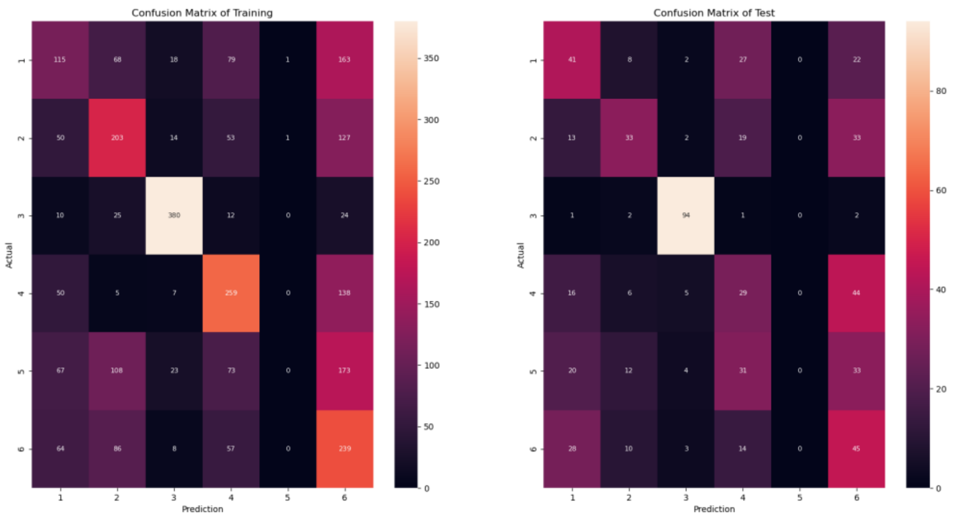

Figure 20: RNN Confusion Matrixes of RNN

From the figure 19, it is seen that loss lines are unstable. Rather, it is expected to see a convergence in plots to a point, hence successful learning is achieved. Despite the fluctuations, the test accuracy is measured as 40.1% percent. Moreover, from the confusion matrix graph in figure 20, we can see that it is not diagonal which is the perfect case. Algorithm guesses the class 3 good and 5 is the worst, but other classes are not as good as 3 because of the biases of the system. As expected, confusion matrix of training has comparatively better result closer to diagonal. Since, the algorithm is not capable of predicting accurately, it shows that it might not be learning. This can be a result of exploding gradients (under and overflow) due to inherent gradient calculation in RNN. Gated units in part b and c of this question are designed to overcome this kind of behaviors.

The unstable accuracy and loss are due to underflow or overflows in the gradient calculations as previously stated. Hence, the neural network becomes not capable conducting efficient learning by giving NaN (Not a Number) value as an output since it could not calculate it. Since, the data provided contains 150 time-steps the accumulation to high values may be natural. This can be prevented by lowering learning rate, which is instructed as 0.1 in the manual, hence, the weight accumulation lowers which also creates dilemma of slowing the training. With low learning rate the fluctuations can disappear, however, loss may not decrease as wanted and training would complete without reaching minima. Since LSTM and GRU overcomes this problem with their gated structure, it will be discussed in next sections. Furthermore, the training is stopped at 30th epoch due to convergence in last eight epochs of validation loss with new loss value in interval of -0.15 and +0.15.

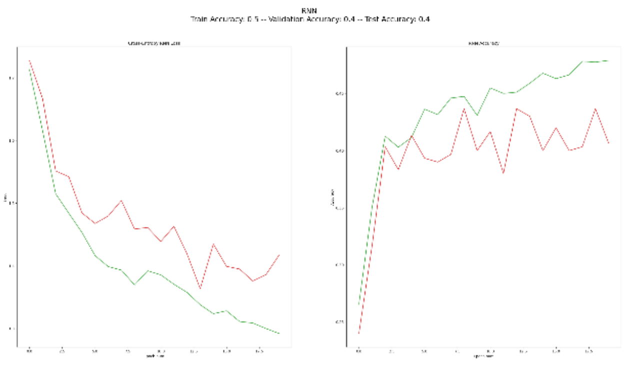

Figure 21: RNN Accuracy Loss with Learning Rate of 0.01

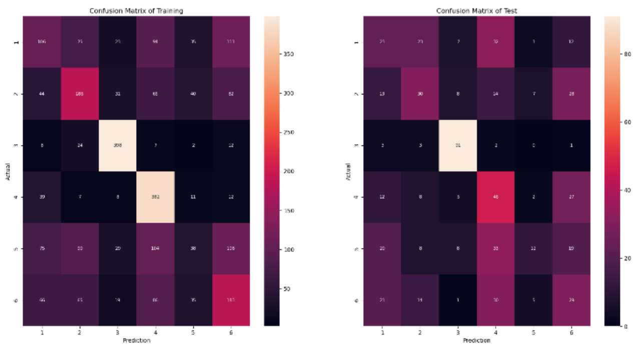

Figure 22: RNN Confusion Matrixes with Learning Rate of 0.01

It is observed that with lower learning rate the fluctuations also decreased, however test accuracy also decreased a little to 38.5% for RNN which also converged around 20th epoch.

b) Long-Short Term Memory (LSTM)

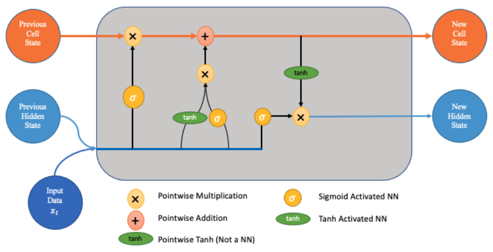

Figure 23: LSTM Architecture [3]

As the schematic of Long-Short Term Memory (LSTM) unit is given figure 21, it is designed with three gates which controls memory used. First part is input gate, and it is responsible for determining the amount of input to be added to existing memory. In addition, second gate s the forget gate responsible for removing (forgetting) data of old-time instants from existing memory. Hence, the complexity lowers. Then, third gate which is output, uses updated cell vector, output of previous step and input and layer activation is founded respectively. Hence, by updating memory constantly, BPTT is kept efficiently by putting some limits which reduces exploding gradients. Forward propagation for each cell is as below where 'c' cell state vector, 'h' hidden activation, 'f' means forget, 'I' means input, 'o' means output;

$$f(t) = sigmoid(W_f \times [h(t-1)x(t)] + b_f$$

$$i(t) = sigmoid(W_i \times [h(t-1)x(t)] + b_i$$

$$\tilde{c}(t) = sigmoid(W_c \times [h(t-1)x(t)] + b_c$$

$$o(t) = sigmoid(W_o \times [h(t-1)x(t)] + b_o$$

$$c(t) = f(t) \times c(t-1) + i(t) \times \tilde{c}(t)$$

$$h(t) = o(t) \times \tanh\,[c(t)]$$

Then, the LSTM network is implemented and trained with the previously mention (given by the manual) parameters and hidden size of 16,32 for multi-layer perceptron.

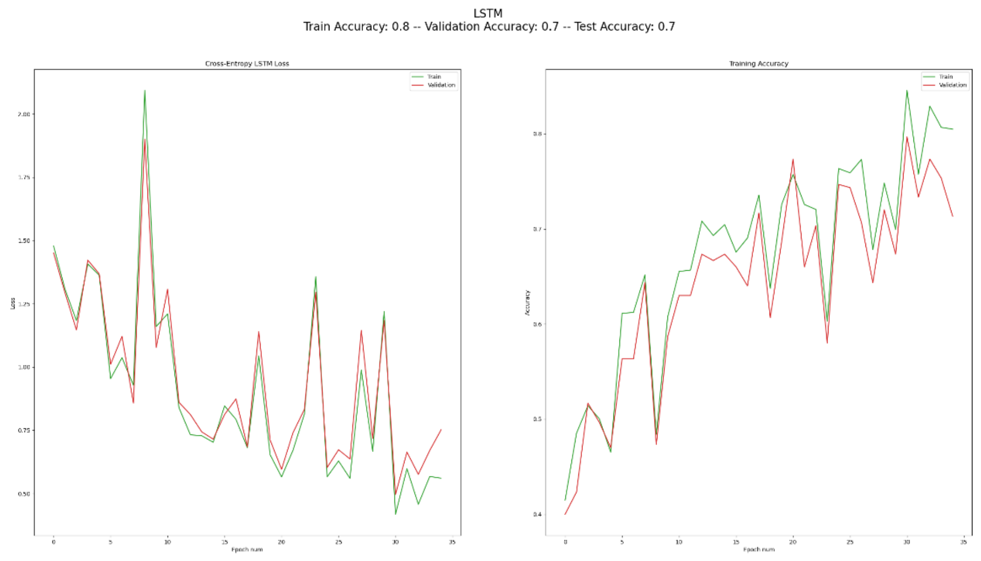

Figure 24: Long-Short Term Memory Losses with Accuracy Test Accuracy: 69.9% (η=0.1,α=0.85,50 epoch, mini batch size of 32)

From figure 22, it is seen that even though there are some fluctuations in loss and accuracy curves, the loss line decreases to appoint while accuracy line increases to 70 percent accuracies (40% in RRN case). If the LSTM and RNN to be compared from their loss and accuracy graphs, LSTM shows a big improvement.

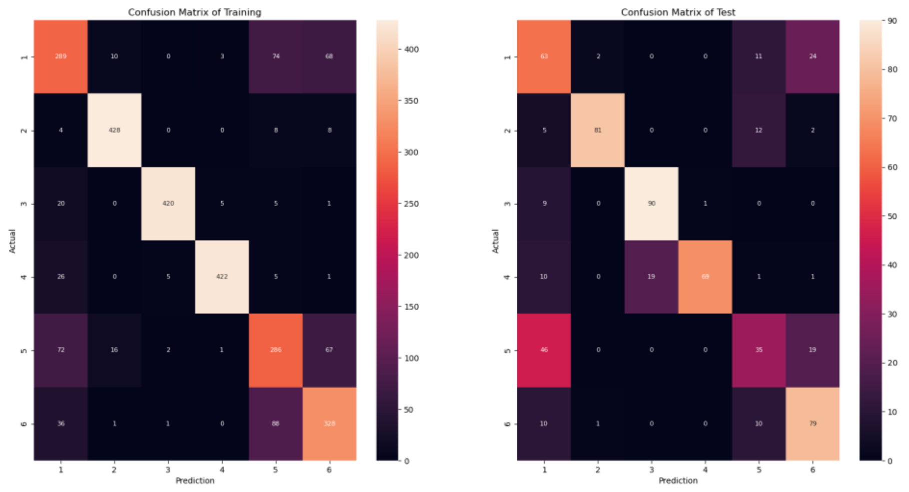

Figure 25: Confusion Matrixes of LSTM

In addition, from confusion matrixes, it is seen that its much closer to diagonal matrixes. Hence, the algorithm more efficient and has higher accuracy so that capable of predicting correctly in most of the times. Similar to RNN result, it is predicting class 3 most correctly and class 5 and 6 with worst accuracy. When the learning rate is decreased fluctuations also decreases but test accuracy also decreases as I saw from different learning rate experiments. With low learning rate, exploding gradients decreasing since less parameter accumulates. But RNN is much less capable of learning with high learning rate values. Hence, it is possible to say LSTM reduces exploding gradient problem with similar learning rates.

c) Gated Recurrent Unit (GRU)

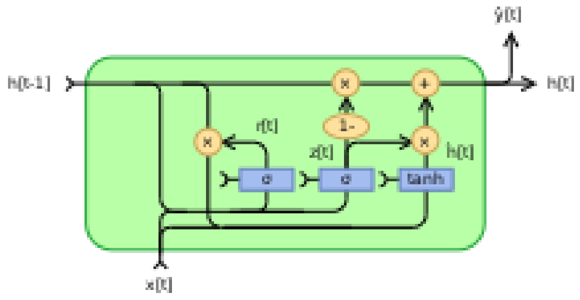

Figure 26: GRU Architecture [4]

In part c, gated recurrent unit (GRU) is implemented. GRU is the modified version of LSTM which has less gates. Hence, it is more simplistic than LSTM results in higher computational efficiency. Since LSTM has more complex structure which is expected to take more time during learning and their overall accuracy and efficiency comparison is ongoing debate. Overall, structure is illustrated in figure 26. GRU contains two gates. These gates are update and reset gates. With sigmoidal activation function first gate takes current input and activation of previous hidden. Second gate, reset is for determining the data used, basically forgets some of past inputs similar to the LSTM forget. Then, output is determined with gate signals. Its mathematical expressions as below;

$$z_t = sigmoid(W_{zh} \times h_{t-1} + W_{zx} \times x_t)$$

$$r_t = sigmoid(W_{rh} \times h_{t-1} + W_{rx} \times x_t)$$

$$\tilde{h}_t = sigmoid(W_h \times (r_t \times h_{t-1}) + W_x \times x_t)$$

$$h_t = z_t \times h_{t-1} + (1 - z_t) \times \tilde{h}_t$$

Then, after this it passes through forward propagation and BPTT is determined with gate signals. Even though, GRU is simpler than LSTM, it reaches very good accuracies.

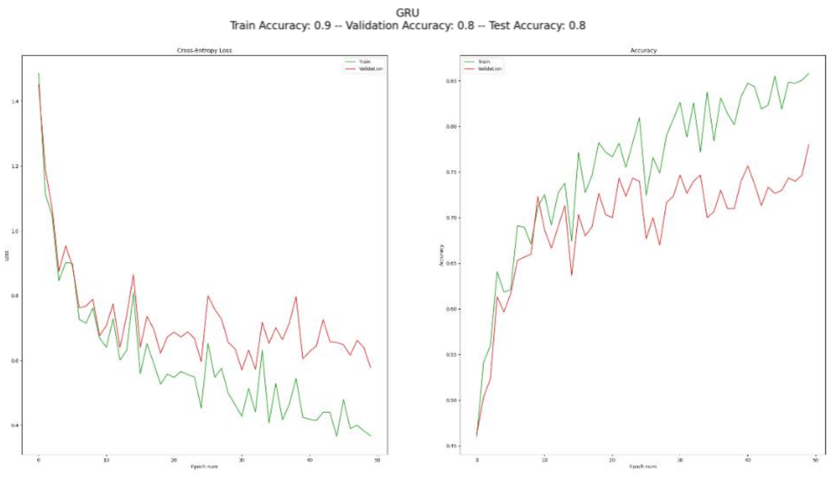

Figure 27: Gated Recurrent Unit Losses with Accuracy Test Accuracy: 75.3% (η=0.1,α=0.85,50 epoch, mini batch size of 32)

Test results shows LSTM and GRU has very close accuracies, however, the GRU algorithm reached these accuracies much faster than LSTM and according to result with lower fluctuations (around five percent test accuracy difference). It takes smaller time to train GRU than LSTM. But both have different advantages and perform better with different tasks. In addition, with less complexity GRU takes less memory (which also results in shorter training time) due to additional gate in LSTM. However, LSTM is more stable to exploding gradient problem and capable of working with higher learning rates due to its complexity, but GRU also reduce effect of gradient exploding.

Other than that, GRU also performed desired performance, so that its confusion matrixes are close to diagonal. Again, it performed good in predicting class 3 and worst in 5 from figure 28.

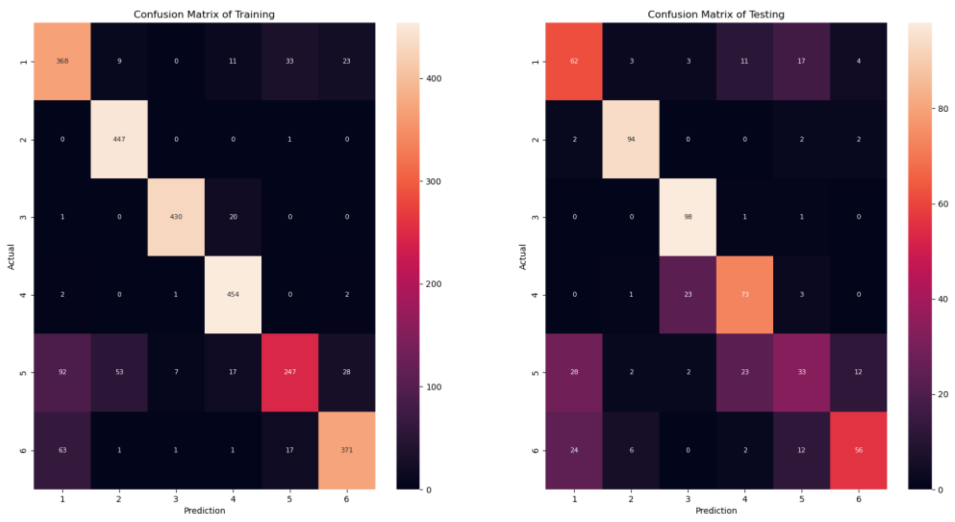

Figure 28: GRU Confusion Matrixes

As overall, result comparing time they consume during training, their accuracy and memory efficiency difference among three neural networks, best is the GRU from the observations. When the data has more complexity (more than 150-time steps) LSTM probably would outperform other neural networks.

Reference

[1] "Basic architecture of a single layer autoencoder made of an encoder ..." [Online]. Available: https://www.researchgate.net/figure/Basic-architecture-of-a-single-layer-autoencoder-made-of-an-encoder-going-from-the-input_fig3_333038461. [Accessed: 17-Dec-2022].

[2] "Recurrent neural networks cheatsheet star," CS 230 - Recurrent Neural Networks Cheatsheet. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. [Accessed: 17-Dec-2022].

[3] R. Dolphin, "LSTM networks: A detailed explanation," Medium, 12-Dec-2021. [Online]. Available: https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9. [Accessed: 17-Dec-2022].

[4] “Gated Recurrent Unit,” Wikipedia, 24-Nov-2022. [Online]. Available: https://en.wikipedia.org/wiki/Gated_recurrent_unit. [Accessed: 17-Dec-2022].

Appendix

Python Code:

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

#########################Q1a####################################3
def rgbtoGray(data1):
    shape=np.shape(data1)
    X=np.zeros([shape[0],1,shape[2],shape[3]])
    X[:,0]=data1[:,0]*0.2126+data1[:,1]*0.7152+data1[:,2]*0.0722
    X=X[:,0,:,:]
    return X

def normalizeIm(X):
    # rangE=[0.1,0.9]
    Y= np.zeros(np.shape(X))
    for i, value in enumerate(X):
        Y[i]=X[i]-np.mean(value)
    std=np.std(X)
    Y=np.clip(Y,std*(-3),std*3)
    Y=(Y - Y.min())/(Y.max() - Y.min())
    Y = 0.1 + Y * 0.8  # map to 0.1 - 0.9
    # Y=np.interp(Y,[std*(-3),std*3],rangE)
    return Y

def normalize(x):
    min_x=x.min()
    max_x=x.max()
    return (x-min_x)/max_x-min_x

def Q1():
    data1= h5py.File("data1.h5", 'r')
    data=data1.get('data')
    data_f=np.array(data)   #data in numpy array for plotting

    data=rgbtoGray(data)
    data=normalizeIm(data)

    data1.close()

    pixel = data.shape[1]
    data_t = np.reshape(data, (data.shape[0], pixel ** 2)) #flatten for the
training part


    data_f = data_f.transpose((0, 2, 3, 1))
    rgb_im, ind0 = plt.subplots(10, 20, figsize=(20, 10))
    gray_im, ind1 = plt.subplots(10, 20, figsize=(20, 10), dpi=200,
facecolor='w', edgecolor='k')

    im_number=np.random.randint(0,10240,size=(10,20))

    for i in range(10):
        for j,k in enumerate(im_number[i]):
            ind0[i, j].imshow(data_f[k].astype('float'))
            ind0[i, j].axis("off")

            ind1[i, j].imshow(data[k], cmap='gray')
            ind1[i, j].axis("off")


    rgb_im.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0,
top=1)
    gray_im.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0,
top=1)
    rgb_im.savefig("200rgb.png")
    gray_im.savefig("200gray.png")
    plt.close("all")

    plt.figure(figsize=(16,16))
    for i,k in enumerate(im_number):
        plt.subplot(10,20,i+1)
        plt.axis('off')
        plt.imshow(data[k[0]],cmap='gray')
    plt.figure(figsize=(16,16))

    for i,k in enumerate(im_number):
        plt.subplot(10,20,i+1)
        plt.axis('off')
        plt.imshow(data_f[k[0]])
    ###########################Q1a############################################
######
    #Determined by expeirment
    lr = 0.08
    epoch=75
    rho = 0.15
    beta = 2.12

    #Q1b these are determined by the manual
    Lambda = 5e-4
```

```python
    Lin = data.shape[2]*data.shape[1]
    Lhid = 64

    params = {"rho": rho, "beta": beta, "Lambda": Lambda, "Lin": Lin, "Lhid":
Lhid}
    ae = AutoEncoder(params,data_t)
    W, Js0 = ae.train(data_t,lr, epoch, batchS = 16)
    W1,_,_,_ = W
    Wnorm = normalize(W1)
    fig = plt.figure(figsize=(100, 70))
    for i in range(64):
        fig.add_subplot(8,8,i+1)
        plt.imshow(Wnorm[:,i].reshape((16,16)),cmap='gray')
    fig.savefig("first.png")



    ######### low-med-high
    # low med high
    cvJs=[]
    index=0
    Lambda = [0, 4e-4, 15e-4]
    Lhid = [36, 49, 81]
    for l in Lambda:
        for h in Lhid:
            s = np.sqrt(h)
            s = int(s)
            print('λ='+str(l)+ ', L='+str(h))
            params = {"rho": rho, "beta": beta, "Lambda": l, "Lin": Lin,
"Lhid": h}
            ae = AutoEncoder(params,data_t)
            W, Js = ae.train(data_t,lr,  epoch, batchS = 16)
            cvJs.append(Js)
            W1,_,_,_ = W
            Wnorm = normalize(W1)
            plt.figure()
            fig = plt.figure(figsize=(100, 70))
            fig.suptitle("lambda = {}, Lhid = {}".format(l, h), fontsize=30)
            for i in range(h):
                fig.add_subplot(s,s,i+1)
                plt.imshow(Wnorm[:,i].reshape((16,16)),cmap='gray')
            fig.savefig("first"+str(index)+".png")
            index+=1
    Lhids = [36, 49, 81]
    lambdas=[0, 4e-4, 15e-4]
    fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Cross Validation Loss", fontsize=20)
    plt.plot(Js0, label='λ='+str(5e-4)+' L=64')
    a=0
    for k in lambdas:
        for i in Lhids:
#            for j in cvJs:
            plt.plot(cvJs[a], label='λ='+str(k)+', L='+str(i))
            plt.legend()
            plt.title("")
            plt.xlabel("Epoch")
            plt.ylabel("Loss")
            a+=1
    plt.savefig("CVLost.png")

######################  End of
Q1  #################################################

#########################Q2###################################################
######
def q2():
    dir_= 'data2.h5'
    data2= h5py.File(dir_, 'r')
    testd=np.array(data2.get('testd'))
    testx=np.array(data2.get('testx'))
    traind=np.array(data2.get('traind'))
    trainx=np.array(data2.get('trainx'))
    vald=np.array(data2.get('vald'))
    valx=np.array(data2.get('valx'))
    words=np.array(data2.get('words'))

    data2.close()

    traind = np.reshape(traind, (traind.shape[0], 1))
    vald = np.reshape(vald, (vald.shape[0], 1))
    testd = np.reshape(testd, (testd.shape[0], 1))
    words = np.reshape(words, (words.shape[0], 1))

    ########## Part a ###############

    lr = 0.15
    mom = 0.85
    epoch = 50
    batchS = 200

    # 8 , 64 ##################
    d2,p2=[8,64]
    sizeL2=[750, d2,p2,250]
    lenL = len(sizeL2) - 1


    fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Train and Validation Lost", fontsize=20)
```

```python
nlpnet2 = nlp(sizeL2, lenL)
out2 = nlpnet2.train(trainx, traind, valx, vald, lr, epoch, batchS, mom)
lostTrainL2, lostValL2, accTrainL2, accValL2 = out2.values()


plt.plot(lostTrainL2, "C0", label="Train Loss")
plt.plot(lostValL2, "C3", label="Validation Loss")
plt.legend()
plt.title("(D, P) = 8, 64")
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")
plt.savefig("question2Lost.png")


# 16, 128 ################
d1,p1=[16,128]
sizeL1=[750, d1,p1,250]

lenL = len(sizeL1) - 1

nlpnet1 = nlp(sizeL1, lenL)
out1 = nlpnet1.train(trainx, traind, valx, vald, lr, epoch, batchS, mom)
lostTrainL1, lostValL1, accTrainL1, accValL1 = out1.values()

fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle("Train and Validation Lost", fontsize=20)

#plt.subplot(1, 3, 2)
plt.plot(lostTrainL1, "C0", label="Train Loss")
plt.plot(lostValL1, "C3", label="Validation Loss")
plt.legend()
plt.title("(D, P) = 16, 128")
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")
plt.savefig("question2Lost2.png")

# 32, 256 ###########
d,p=[32,256]
sizeL=[750, d,p,250]

lenL = len(sizeL1) - 1


nlpnet = nlp(sizeL, lenL)
out = nlpnet.train(trainx, traind, valx, vald, lr, epoch, batchS, mom)
lostTrainL, lostValL, accTrainL, accValL = out.values()

fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
```

```
    fig.suptitle("Train and Validation Lost", fontsize=20)

    plt.plot(lostTrainL, "C0", label="Train Loss")
    plt.plot(lostValL, "C3", label="Validation Loss")
    plt.legend()
    plt.title("(D, P) = (32, 256)")
    plt.xlabel("Epoch")
    plt.ylabel("Cross-Entropy Loss")

    plt.savefig("question2Lost3.png")



    fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Train and Validation Accuracy", fontsize=20)

    plt.plot(accTrainL2, "C0", label="Train Accuracy")
    plt.plot(accValL2, "C3", label="Validation Accuracy")
    plt.legend()
    plt.title("(D, P) = (8, 64)")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.savefig("question2Accuracy.png")

    fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Train and Validation Accuracy", fontsize=20)

    plt.plot(accTrainL1, "C0", label="Train Accuracy")
    plt.plot(accValL1, "C3", label="Validation Accuracy")
    plt.legend()
    plt.title("(D, P) = (16, 128)")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.savefig("question2Accuracy2.png")

    fig = plt.figure(figsize=(30, 10), dpi=80, facecolor='w', edgecolor='k')
    fig.suptitle("Train and Validation Accuracy", fontsize=20)

    plt.plot(accTrainL, "C0", label="Train Accuracy")
    plt.plot(accValL, "C3", label="Validation Accuracy")
    plt.legend()
    plt.title("(D, P) = (32, 256)")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")

    plt.savefig("question2Accuracy3.png")
    ######### b  ###################
    temp = testx - 1
    data0 = np.zeros((temp.shape[0], 0))
    for i in range(temp.shape[1]):
        temp2 = np.zeros((temp.shape[0], 250))
        temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
        data0 = np.hstack((data0, temp2))


    testClas,_ = nlpnet.choice(data0)

    print("\n\nTest Accuracy of (32, 256): ", (testClas == testd).mean() )

    ind = 10
    np.random.seed(666)
    p = np.random.permutation(testx.shape[0])
    testx = testx[p][:ind]
    testd = testd[p][:ind]
    ##################################################
    temp = testx - 1
    data1 = np.zeros((temp.shape[0], 0))
    for i in range(temp.shape[1]):
        temp2 = np.zeros((temp.shape[0], 250))
        temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
        data1 = np.hstack((data1, temp2))

    _,predTest = nlpnet.choice(data1)

    n = 10
    s = (np.argsort(-predTest, axis=1) + 1)[:, :n]

    for i in range(ind):
            print("\n")
            print("Sequence:", words[testx[i][0] - 1], words[testx[i][1] - 1],
words[testx[i][2] - 1])
            print("Label:", words[testd[i] - 1])
            for j in range(n):
                print(str(j + 1) + ". ", words[s[i][j] - 1])
#########################End of Q2
##################################################

#####################
Q3##############################################################
def q3():
    filename = "data3.h5"
    h5 = h5py.File(filename, 'r')
    trX = h5['trX'][()].astype('float64')
    tstX = h5['tstX'][()].astype('float64')
    trY = h5['trY'][()].astype('float64')
    tstY = h5['tstY'][()].astype('float64')
    h5.close()

    mom = 0.85
    lr = 0.1
    epoch = 50 #max 50
    batch_size = 32

    hs = [64,16]
    print("RNN\n")
    rnnQ1 = sRNN(128,hs, modelType='rnn')
    lostTrain, lostVal, accTrain, accVal, trainconf = rnnQ1.train(trX, trY,
lr, mom, batch_size, epoch, modelType='rnn').values()
    accuracy, _, confidence = rnnQ1.choice(tstX, tstY)

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

#
============================================================================
    hs2 = [16,32]
    mom = 0.85
    lr = 0.05
    epoch = 50 #max50
    batch_size = 32
    print("LSTM\n")
    lstmQ2 = sRNN(128, hs=hs2, modelType="lstm")
    lostTrain2, lostVal2, accTrain2, accVal2, trainconf2 = lstmQ2.train(trX,
trY, lr, mom, batch_size,epoch,modelType='lstm').values()
    accuracy2,_,confidence2 = lstmQ2.choice(tstX, tstY)
    print("\nTest Accuracy: ", accuracy2, "\n")

    hs3 = [64,32]
    mom = 0.85
    lr = 0.05
    epoch = 50 #50
    batch_size = 32
    print("GRU\n")
    gruQ3 = sRNN(128, hs=hs3, modelType="gru")
    lostTrain3, lostVal3, accTrain3, accVal3, trainconf3 = gruQ3.train(trX,
trY, lr, mom, batch_size,epoch,modelType='gru').values()
    accuracy3,_,confidence3 = gruQ3.choice(tstX, tstY)
    print("\nTest Accuracy: ", accuracy3, "\n")

#
============================================================================
    #plot the graphs
    fig = plt.figure(figsize=(30, 15))
    fig.suptitle("RNN\n Train Accuracy: {:.1f} -- Validation Accuracy: {:.1f}
-- Test Accuracy: {:.1f}\n "
```

```
    .format(accTrain[-1], accVal[-1], accuracy), fontsize=20)
    plt.subplot(1, 2, 1)
    plt.plot(lostTrain, "C2")

    plt.title("Cross-Entropy RNN Loss")
    plt.xlabel("Epoch num")
    plt.ylabel("Loss")
    plt.plot(lostVal, "C3")

    plt.subplot(1, 2, 2)
    plt.plot(accTrain, "C2")


    plt.title("RNN Accuracy")
    plt.xlabel("Epoch num")
    plt.ylabel("Accuracy")
    plt.plot(accVal, "C3")

    plt.savefig("question3RNNacc.png")

    plt.figure(figsize=(20, 10))


    lbls = [1, 2, 3, 4, 5, 6]
    plt.subplot(1, 2, 1)
    sn.heatmap(trainconf, annot=True, annot_kws={"size": 8}, xticklabels=lbls,
yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')

    plt.title("Confusion Matrix of Training")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.subplot(1, 2, 2)
    sn.heatmap(confidence, annot=True, annot_kws={"size": 8},
xticklabels=lbls, yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')

    plt.title("Confusion Matrix of Test")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.savefig("question3ConfRec.png")


#
=============================================================================
    ###
    fig = plt.figure(figsize=(30, 15))
    fig.suptitle("LSTM\n Train Accuracy: {:.1f} -- Validation Accuracy: {:.1f}
-- Test Accuracy: {:.1f}\n "
```

```
    .format( accTrain2[-1], accVal2[-1], accuracy2), fontsize=20)
    plt.subplot(1, 2, 1)
    plt.plot(lostTrain2, "C2", label="Train")
    plt.legend()

    plt.title("Cross-Entropy LSTM Loss")
    plt.xlabel("Epoch num")
    plt.ylabel("Loss")
    plt.plot(lostVal2, "C3", label="Validation")
    plt.legend()


    plt.subplot(1, 2, 2)
    plt.plot(accTrain2, "C2",label="Train")
    plt.legend()


    plt.title("Training Accuracy")
    plt.xlabel("Epoch num")
    plt.ylabel("Accuracy")
    plt.plot(accVal2, "C3", label="Validation")
    plt.legend()

    plt.savefig("question3LSTM.png")
    plt.figure(figsize=(20, 10))

    lbls = [1, 2, 3, 4, 5, 6]
    plt.subplot(1, 2, 1)
    sn.heatmap(trainconf2, annot=True, annot_kws={"size": 8},
xticklabels=lbls, yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')

    plt.title("Confusion Matrix of Training")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.subplot(1, 2, 2)
    sn.heatmap(confidence2, annot=True, annot_kws={"size": 8},
xticklabels=lbls, yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')
    plt.title("Confusion Matrix of Test")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.savefig("question3ConfLSTM.png")




    ###
    fig = plt.figure(figsize=(30, 15))
    fig.suptitle("GRU\nTrain Accuracy: {:.1f} -- Validation Accuracy: {:.1f} -
- Test Accuracy: {:.1f}\n "
    .format( accTrain3[-1], accVal3[-1], accuracy3), fontsize=20)
    plt.subplot(1, 2, 1)
    plt.plot(lostTrain3, "C2", label='Train')
    plt.legend()

    plt.title("Cross-Entropy Loss")
    plt.xlabel("Epoch num")
    plt.ylabel("Loss")
    plt.plot(lostVal3, "C3", label='Validation')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(accTrain3, "C2", label='Train')
    plt.legend()

    plt.title(" Accuracy")
    plt.xlabel("Epoch num")
    plt.ylabel("Accuracy")
    plt.plot(accVal3, "C3", label='Validation')
    plt.legend()


    plt.savefig("question3GRU.png")
    plt.figure(figsize=(20, 10))

    lbls = [1, 2, 3, 4, 5, 6]
    plt.subplot(1, 2, 1)
    sn.heatmap(trainconf3, annot=True, annot_kws={"size": 8},
xticklabels=lbls, yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')

    plt.title('Confusion Matrix of Training')
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.subplot(1, 2, 2)
    sn.heatmap(confidence3, annot=True, annot_kws={"size": 8},
xticklabels=lbls, yticklabels=lbls,
    cmap=sn.cm.rocket, fmt='g')

    plt.title("Confusion Matrix of Testing")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.savefig("question3ConfGRU.png")
#
=============================================================================
########################End of
Q3#####################################################

##########Q1 Calss###############
class AutoEncoder(object):
    def __init__(self,params,data):
        Lin = params["Lin"]
        Lhid = params["Lhid"]
        self.rho = params["rho"]
        self.beta = params["beta"]
        self.Lambda = params["Lambda"]


        Lout = Lin # autoencoder takes like this
        L = Lin + Lhid

        w_0 = np.sqrt(6/L)
        W_1 = np.random.uniform(-1*w_0,w_0,size=(Lin, Lhid))
        b_1 = np.random.uniform(-1*w_0,w_0,size=(1,Lhid))

        W_2 = W_1.T
        b_2 = np.random.uniform(-1*w_0,w_0,size=(1, Lout))

        self.We = [W_1, W_2, b_1, b_2]

    def aeCost(self,We,data):
        W_1,W_2,b_1,b_2=We
        N=data.shape[0]

        update=np.dot(data,W_1)+b_1
        result=self.sigmoid(update)
        update2=np.dot(result,W_2)+b_2
        output=self.sigmoid(update2)

        dloss = -(data-output)/N
        dtykh_1 = self.Lambda*W_1
        dtykh_2 = self.Lambda*W_2

        #the coss function in three parts (instructed in the manual)

        loss = 0.5/N * (np.linalg.norm(data - output, axis=1) ** 2).sum()
        tykh = 0.5 * self.Lambda * (np.sum(W_1 ** 2) + np.sum(W_2 ** 2))
        rho_hat = result.mean(axis=0, keepdims=True)
        KLdiv = self.rho * np.log(self.rho/rho_hat) + (1 - self.rho) *
np.log((1 - self.rho)/(1 - rho_hat))
        KLdiv= self.beta*KLdiv.sum()

        dKLdiv = self.beta * (- self.rho/rho_hat + (1-self.rho)/(1 -
rho_hat))/N
```

```
        #sum of three parts is the loss
        J=loss+tykh+KLdiv
        cache={'data':data,'result':result,'output':output}
        J_grad={'dloss':dloss, 'dtykh_1':dtykh_1, 'dtykh_2': dtykh_2,
'dKLdiv':dKLdiv}

        return J_grad,cache,J

    def train(self,data,learning_rate,epoch,batchS):
        N=data.shape[0]
        Js=[]
        iteration = round(N/batchS)
        We=self.We
        for i in range(epoch):
            totJ = 0
            indS = 0
            indE = batchS

            randomChoice = np.random.permutation(N)
            randomData = data[randomChoice]

            mWe = (0, 0, 0, 0)

            for j in range(iteration):

                batch = randomData[indS:indE]

                J_grad, cache, J = self.aeCost(We,batch)
                We=self.GDsolver(We, J_grad, cache, learning_rate)

                totJ = totJ+ J
                indS = indE
                indE += batchS

            totJ /=iteration
            print("Loss: {:.2f}, Epoch {} out of {}".format(totJ, i+1, epoch))
            Js.append(totJ)
        print("\n")
        return We,Js

    def learn(self,learning_rate,grads,We):
        grads = grads*learning_rate
        We[0]-=grads[0]
        We[1]-=grads[1]
        We[2]-=grads[2]
        We[3]-=grads[3]
        return We
```

```python
    def GDsolver(self,We,J_grad,cache,learning_rate):
        # dW_1,dW_2,db_1,db_2= [0, 0, 0, 0]
        data=cache['data']
        hidden=cache['result']
        dHidden=hidden-hidden**2
        output=cache['output']
        dOutput=output-output**2

        dloss=J_grad['dloss']
        dtykh_1=J_grad['dtykh_1']
        dtykh_2=J_grad['dtykh_2']
        dKLdiv=J_grad['dKLdiv']

        change=dloss*dOutput

        W_2=np.dot(hidden.T,change)+dtykh_2
        db_2=np.sum(change,axis=0,keepdims=True)

        change1 = dHidden * (np.dot(change,We[1].T) + dKLdiv)

        W_1 = np.dot(data.T, change1) + dtykh_1
        db_1 = np.sum(change1,axis=0, keepdims=True)

        dw_2=(1/2)*(W_1.T + W_2)
        dw_1= W_2.T

        grads=np.array([dw_1,dw_2,db_1,db_2])
        We=self.learn(learning_rate,grads,We)
        return We

    def softmax(self,x):
        soft=np.zeros(1,len(x))
        for i in len(x):
            soft[i]=np.exp(x[i])/sum(np.exp(x))
        return soft

    def sigmoid(self,x):
        return 1/(1+np.exp(-x))

    def relu(self,X):
        return np.maximum(0,X)
##########Q2 Class################
class nlp(object):
    def __init__(self, sizeL, lenL = 2, std = 0.01, seed=666):
        #sizeL= all layer len
        # lenL=layer length,
        self.seed = seed
        Ws = []
        bs= []
        for i in range(lenL):
            if i==0:
                np.random.seed(self.seed)
                a = np.random.normal(0, std, size=(int(sizeL[i]/3), sizeL[i +
1]))
                Ws.append(np.vstack((a, a, a)))
                np.random.seed(self.seed)
                bs.append(np.zeros((1, sizeL[i + 1])))
                continue
            np.random.seed(self.seed)
            Ws.append(np.random.normal(0, std, size=(sizeL[i], sizeL[i + 1])))
            np.random.seed(self.seed)
            bs.append(np.random.normal(0, std, size=(1, sizeL[i + 1])))

        self.mom = {"Ws": [None] * lenL, "bs": [None] * lenL}
        self.sizeL = sizeL
        self.param = {"Ws": Ws, "bs": bs}
        self.lenL = lenL
        self.activ = ["sigmoid"] * (lenL - 1) + ["softmax"]

    def cost(self, data, gt):
        param=self.param
        Ws,bs =param.values()
        activ = self.activ
        dataL = [data]
        grads = [1]
        batchS = data.shape[0]
        lenL=self.lenL

        for i in range(lenL):
            post = np.dot(dataL[i], Ws[i]) + bs[i]
            out, grad = self.activition( post,activ[i])
            dataL.append(out)
            grads.append(grad)

        guess = dataL[-1]
        cost = self.CE(gt, guess)
        dE = guess
        dE[gt == 1]-= 1
        dE = dE/batchS

        dWs = []
        dbs = []
        identity = np.ones((1, batchS))

        for i in reversed(range(lenL)):
            dWs.append(np.dot(dataL[i].T , dE))
            dbs.append(np.dot(identity , dE))
            dE = grads[i] * (np.dot(dE , Ws[i].T))

        return cost, {'dWs': dWs[::-1], 'dbs': dbs[::-1]}

    def CE(self, expect, result):
        log=np.log(result)
        return np.sum(-1*expect* log)/ expect.shape[0]

    def activition(self,data,activation):
        if activation=='softmax':
            Z=np.exp(data)/np.sum(np.exp(data),axis=1,keepdims=True)
            d=None
        elif activation=='sigmoid':
            Z=1/(1+np.exp(-data))
            d=Z-Z**2
        elif activation=='relu':
            #Z=np.max(0.0,data)
            Z = data * (data > 0)
            d=1*(data>0)
        elif activation=='tanh':
            Z=np.tanh(data)
            d=1- Z**2
        return Z,d

    def choice(self, data):
        Ws = self.param["Ws"]
        bs = self.param["bs"]
        activ = self.activ
        lenL=self.lenL
        dataL = [data]

        for i in range(lenL):
            post = np.dot(dataL[i], Ws[i]) + bs[i]
            dataL.append(self.activition(post,activ[i])[0])
        # three word embedding one for each word
        out = (np.argmax(dataL[-1], axis=1) + 1).T
        out = np.reshape(out, (out.shape[0], 1))
        return out,dataL[-1]

    def train(self, data, gt, valData, valGt, lr=0.2,
epoch=50,batchS=100,mom=0):
        lostTrainL = []
        lostValL = []
        accTrainL = []
        accValL = []

        param=self.param
        iteration = int(data.shape[0] / batchS)
        lenL=self.lenL
        sizeL=self.sizeL
        moms=self.mom

        #one hot encode word out of 250 words so the matrix sghows the index 
of corresponding word
        temp = data - 1
        data0 = np.zeros((temp.shape[0], 0))
        for i in range(temp.shape[1]):
            temp2 = np.zeros((temp.shape[0], 250))
            temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
            data0 = np.hstack((data0, temp2))

        temp = gt - 1
        gt0 = np.zeros((temp.shape[0], 0))
        for i in range(temp.shape[1]):
            temp2 = np.zeros((temp.shape[0], 250))
            temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
            gt0 = np.hstack((gt0, temp2))

        temp = valData - 1
        valDAta0 = np.zeros((temp.shape[0], 0))
        for i in range(temp.shape[1]):
            temp2 = np.zeros((temp.shape[0], 250))
            temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
            valDAta0 = np.hstack((valDAta0, temp2))

        temp = valGt - 1
        valGt0 = np.zeros((temp.shape[0], 0))
        for i in range(temp.shape[1]):
            temp2 = np.zeros((temp.shape[0], 250))
            temp2[np.arange(temp.shape[0]), temp[:, i]] = 1
            valGt0 = np.hstack((valGt0, temp2))

        for i in range(epoch):

            np.random.seed(self.seed)
            shuff = np.random.permutation(data0.shape[0])
            data0 = data0[shuff]
            gt0 = gt0[shuff]
            gt = gt[shuff]

            for e in range(lenL):
                self.mom["Ws"][e] = np.zeros((sizeL[e],sizeL[e+1]))
                self.mom["bs"][e] = np.zeros((1, sizeL[e+1]))

            first = 0
            last = batchS
            lostTrain = 0
            for j in range(iteration):
                batchTr0 = data0[first:last]
                batchGt0 = gt0[first:last]
                batchGt = gt[first:last]

                cost, grad = self.cost(batchTr0, batchGt0)
                lostTrain += cost

                for k in range(lenL):

                    if k == 0:
                        moms["Ws"][k] = lr * (grad["dWs"][k] + mom *
moms["Ws"][k])
                        dword1, dword2, dword3 = np.array_split(moms["Ws"][k],
3, axis=0)
                        dWord = (dword1 + dword2 + dword3)/3
                        param["Ws"][k] -= np.vstack((dWord, dWord, dWord))


                    moms["Ws"][k] = lr * (grad["dWs"][k] + mom *
moms["Ws"][k])
                    moms["bs"][k] = lr * (grad["dbs"][k] + mom *
moms["bs"][k])
                    param["Ws"][k] = param["Ws"][k]-moms["Ws"][k]
                    param["bs"][k] = param["bs"][k]-moms["bs"][k]

                first = last
                last += batchS

            choiceT,_ = self.choice(data0)
            accuracyTr = (choiceT == gt).mean()

            choiceV,_ = self.choice(valDAta0)
            accuracyV = (choiceV == valGt).mean()

            _,val_pred = self.choice(valDAta0)
            val_loss = self.CE(valGt0, val_pred)


            print('\r(D, P) = (%d, %d). loos Train: %f, loss Val: %f, accuracy
Train: %f, accuracy Val: %f [%d out of %d].'
                    % (sizeL[1], sizeL[2], lostTrain/(j+1), val_loss,
accuracyTr, accuracyV, i + 1, epoch))

            lostTrainL.append(lostTrain/iteration)
            lostValL.append(val_loss)
            accTrainL.append(accuracyTr)
            accValL.append(accuracyV)
            if i > 15 :
                check = lostValL[-15:]
                check = sum(check) / len(check)

                lim = 0.02

                if (check - lim) < val_loss < (check + lim) and val_loss <
3.5:
                    print(" Training is stopped due to cross entropy
convergence in Validation ")
                    return {'lostTrainL': lostTrainL, 'lostValL': lostValL,
                            'accTrainL': accTrainL, 'accValL': accValL}
        return {'lostTrainL': lostTrainL, 'lostValL': lostValL,
                            'accTrainL': accTrainL, 'accValL': accValL}
###########End#################
##########Q3 Class###############
class sRNN(object):

    def __init__(self, n_number,hs, modelType ,outNum=6, inputs=3):
        #train=(3000, 150,3) sample,series,sensor
        #neuron number= 128
        self.modelType0=modelType
        trainS=([inputs,n_number]+hs)+[outNum]
        self.Size=trainS
        self.Layer_len=len(trainS)-1

        hs=trainS[1]

        if modelType=='rnn':
            n=inputs+hs
            x=np.sqrt(6/n)
            W_ih=np.random.uniform(-x, x, size=(inputs, hs))
            x = np.sqrt(6 / (2*hs))
            W_hh = np.random.uniform(-x, x, size=(hs, hs))
            b = np.zeros((1, hs))
            self.We = {"Wih": W_ih, "Whh": W_hh, "b": b}
            self.inLayer = {"Wih": 0, "Whh": 0, "b": 0}

        elif modelType=='lstm':
            n=inputs+2*hs
            X=np.sqrt(6/n)
            Wini = np.random.uniform(-X, X, size=(n-hs, hs))
            Win = np.random.uniform(-X, X, size=(n-hs, hs))
            Wsec = np.random.uniform(-X, X, size=(n-hs, hs))
            Wout = np.random.uniform(-X, X, size=(n-hs, hs))

            bini = np.zeros((1, hs))
            bin1 = np.zeros((1, hs))
            bsec = np.zeros((1, hs))
            bout = np.zeros((1, hs))

            self.We = {"Wini": Wini, "bini": bini,
                      "Win": Win, "bin1": bin1,
                      "Wsec": Wsec, "bsec": bsec,
                      "Wout": Wout, "bout": bout}
            self.inLayer = {"Wini": 0, "bini": 0,
                            "Win": 0, "bin1": 0,
                            "Wsec": 0, "bsec": 0,
                            "Wout": 0, "bout": 0}
        elif modelType=='gru':
            Nih = np.sqrt(6 / (inputs + hs))
            Nhh = np.sqrt(6 / (2* hs))

            Wa = np.random.uniform(-Nih, Nih, size=(inputs, hs))
            Ta = np.random.uniform(-Nhh, Nhh, size=(hs, hs))
            ba = np.zeros((1, hs))

            Wb = np.random.uniform(-Nih, Nih, size=(inputs, hs))
            Tb = np.random.uniform(-Nhh, Nhh, size=(hs, hs))
            bb = np.zeros((1, hs))

            Wc = np.random.uniform(-Nih, Nih, size=(inputs, hs))
            Tc = np.random.uniform(-Nhh, Nhh, size=(hs, hs))
            bc = np.zeros((1, hs))

            self.We = {"Wa": Wa, "Ta": Ta, "ba": ba,
                      "Wb": Wb, "Tb": Tb, "bb": bb,
                      "Wc": Wc, "Tc": Tc, "bc": bc}
            self.inLayer = {"Wa": 0, "Ta": 0, "ba": 0,
                            "Wb": 0, "Tb": 0, "bb": 0,
                            "Wc": 0, "Tc": 0, "bc": 0}

        wMuLP = []
        bMuLP = []
        for i in range(1, self.Layer_len):
            x = np.sqrt(6 / (trainS[i] + trainS[i+1]))
            wMuLP.append(np.random.uniform(-x, x, size=(trainS[i],
trainS[i+1])))
            bMuLP.append(np.zeros((1, trainS[i+1])))
        self.MLPlayer_len = len(wMuLP)
        self.MLPparam = {"W": wMuLP, "b": bMuLP}
        self.MLPm = {"W": [0] * self.MLPlayer_len, "b": [0] *
self.MLPlayer_len,}

    def forward(self,data, modelType):
        MLPparam=self.MLPparam
        out=[]
        d=[]
        hidden=0
        dHidden=0
        cache=0

        modelType0=self.modelType0

        if modelType0 =='gru':
            hidden,cache=self.forwardGRU(data)
            out.append(hidden)
            d.append(1)
        elif modelType0 =='lstm':
            hidden,cache=self.forwardLSTM(data)
            out.append(hidden)
            d.append(1)
        elif modelType0 =='rnn':
            hidden,dHidden=self.forwardRNN(data)
            out.append(hidden[:, -1, :])
            d.append(dHidden[:, -1, :])

        for i in range(self.MLPlayer_len-1):
            activition, deriv = self.forwardMLP(out[-1], MLPparam["W"][i], 
MLPparam["b"][i], "relu")
            out.append(activition)
            d.append(deriv)

        choice = self.forwardMLP(out[-1], MLPparam["W"][-1], MLPparam["b"][-
1], "softmax")[0]
        return choice,out,d,hidden,dHidden,cache

    def backward(self,data,activition,dMLP,dEp, 
modelType,hidden=None,dhidden=None,cache=None):
        MLPparam=self.MLPparam
        MLPlayer_len=self.MLPlayer_len
        gMLP ={"W": [0] * MLPlayer_len, "b": [0] * MLPlayer_len}
        # back before rnn
        for i in reversed(range(MLPlayer_len)):
            gMLP["W"][i], gMLP["b"][i], dEp = self.backMLP(MLPparam["W"][i], 
activition[i], dMLP[i], dEp)
        # back in time
        if modelType == 'gru':
            gRNN = self.backGRU(data, cache, dEp)
        elif modelType == 'lstm':
            gRNN = self.backLSTM(cache, dEp)
        elif modelType == 'rnn':
            gRNN = self.backRNN(data,hidden, dhidden, dEp)

        return gRNN, gMLP

    def forwardMLP(self,data,W,b,activition):
        MLP=np.dot(data,W)+b
        Z,d=self.activition(MLP,activition)
        return Z,d

    def backMLP(self, w, preout,dPreout,dEpre):
        db=np.sum(dEpre,axis=0,keepdims=True)
        dW=np.dot(preout.T,dEpre)
        dE=dPreout*np.dot(dEpre,w.T) #new delta from old delta and derivative
        return dW,db,dE

    def forwardRNN(self,data):
        numN=self.Size[1] #neuron number
        dims=np.shape(data)
        numS=dims[0] #sample number
        t=dims[1]     #time sequence
        sensor=dims[2]  #three sensor dim

        We=self.We
        Wih=We['Wih']
        Whh=We['Whh']
        b=We['b']

        hiddenL=np.zeros((numS,numN))
        hidden=np.empty((numS,t,numN))
        dHidden=np.copy(hidden)

        for i in range(t):
            d=data[:,i,:]
            a=np.dot(d,Wih)
            b=np.dot(hiddenL,Whh)+b
            hidden[:, i, :],dHidden[:, i, :] = self.activition(a+b, "tanh")
            hiddenL = hidden[:, i, :]
        return hidden,dHidden

    def backRNN(self,data,hidden,dhidden,dEp):
        numN=self.Size[1] #neuron number
        dims=np.shape(data)
        numS=dims[0] #sample number
        t=dims[1]     #time sequence
        sensor=dims[2]  #three sensor dim

        We=self.We
        Whh=We['Whh']
        dW_ih,dW_hh,db=[0,0,0]

        for i in reversed(range(t)):
            batch=data[:,i,:]

            if i>0:
                hiddenL=hidden[:,i-1,:]
                dhiddenL=dhidden[:,i-1,:]
            else:
                hiddenL=np.zeros((numS,numN))  #for first layer
                dhiddenL=0
            dW_ih=dW_ih+np.dot(batch.T,dEp)
            dW_hh=dW_hh+np.dot(hiddenL.T,dEp)
            db=db+np.sum(dEp,axis=0,keepdims=True)
            dEp= np.dot(dEp,Whh)*dhiddenL
        grads={'dWih':dW_ih,'dWhh':dW_hh,
                'db':db, 'dEp':dEp}
        return grads

    def forwardLSTM(self,data):
        #store activation
        We=self.We
        Wini,bini,Win,bin1,Wsec,bsec,Wout,bout=We.values()
        numN=self.Size[1] #neuron number
        dims=np.shape(data)
        numS=dims[0] #sample number
        t=dims[1]    #time sequence
        sensor=dims[2]  #three sensor dim

        hiddenL = np.zeros((numS, numN))
        cL = np.zeros((numS, numN)) #for firrst iter

        k = np.empty((numS, t, sensor + numN))
        c = np.empty((numS, t, numN))

        hiddenf = np.empty((numS, t, numN))
        hiddeni = np.empty((numS, t, numN))
        hiddenz = np.empty((numS, t, numN))
        hiddeno = np.empty((numS, t, numN))
        tanhc = np.empty((numS, t, numN))

        dhiddenf = np.empty((numS, t, numN))
        dhiddeni= np.empty((numS, t, numN))
        dhiddenz = np.empty((numS, t, numN))
        dhiddeno = np.empty((numS, t, numN))
        dtanh = np.empty((numS, t, numN))

        for i in range(t):
            dataTime=data[:,i,:]
            temp=np.hstack((hiddenL,dataTime))
            f,df = self.activition(np.dot(temp, Wini) + bini, "sigmoid")
            i0,di = self.activition(np.dot(temp, Win) + bin1, "sigmoid")
            z,dz = self.activition(np.dot(temp ,Wsec) + bsec, "tanh")
            o,do= self.activition(np.dot(temp, Wout) + bout,
"sigmoid")
```

```
            a = np.multiply(z , i0)
            b= np.multiply(cL , f)
            multC= a+b
            tanh_c,dtan_h=self.activition(multC,'tanh')
            multZ=np.multiply(o,tanh_c)

            cL=multC
            hiddenL=multZ


            hiddenf[:,i,:],dhiddenf[:,i,:] = f,df
            hiddeni[:,i,:],dhiddeni[:,i,:] = i0,di
            hiddenz[:,i,:],dhiddenz[:,i,:] =z,dz
            hiddeno[:,i,:],dhiddeno[:,i,:] =o,do
            tanhc[:,i,:],dtanh[:,i,:] = tanh_c,dtan_h
            k[:,i,:]=temp

        return multZ,
{'k':k,'c':c,'hiddenf':hiddenf,'hiddeni':hiddeni,'hiddenz':hiddenz,'hiddeno':h
iddeno,
                        'dhiddenf':dhiddenf,'dhiddeni':dhiddeni,'dhiddenz':dhi
ddenz,'dhiddeno':dhiddeno,
                        'tanhc':tanhc,'dtanh':dtanh}

    def backLSTM(self,cache,dEu):
        We=self.We
        Wini,_,Win,_,Wsec,_,Wout,_=We.values()
        k,c,hiddenf,hiddeni,hiddenz,hiddeno,dhiddenf,dhiddeni,dhiddenz,dhidden
o,tanhc,dtanh = cache.values()
        numN=self.Size[1] #neuron number
        t=np.shape(k)[1]

        dWini,dbini,dWin,dbin1,dWsec,dbsec,dWout,dbout=0,0,0,0,0,0,0,0
        #gradients start with zero since no grad first

        for r in reversed(range(t)):
            know = k[:, r, :]

            if r>0:
                cLast=c[:,r-1,:]
            elif r<=0:
                cLast=0 #0  for the ebeginning

            delc = dEu * hiddeno[:, r, :] * dtanh[:, r, :]
            delhf = delc * cLast * dhiddenf[:, r, :]
            delhi = delc * hiddenz[:, r, :] * dhiddeni[:, r, :]
            delhz = delc * hiddeni[:, r, :] * dhiddenz[:, r, :]
            delho = dEu * tanhc[:, r, :] * dhiddeno[:, r, :]
            #add to weight's change
            dWini += np.dot(know.T ,delhf)
            dbini += np.sum(delhf,axis=0, keepdims=True)

            dWin += np.dot(know.T , delhi)
            dbin1 += np.sum(delhi,axis=0, keepdims=True)

            dWsec += np.dot(know.T, delhz)
            dbsec += np.sum(delhz,axis=0, keepdims=True)

            dWout +=np.dot(know.T, delho)
            dbout += np.sum(delho,axis=0, keepdims=True)

            #update gradients of gates
            delf=np.dot(delhf, Wini.T[:,:numN])
            deli=np.dot(delhi, Win.T[:,:numN])
            delz=np.dot(delhz, Wsec.T[:,:numN])
            delo=np.dot(delho, Wout.T[:,:numN])

            dEu = delf+deli+delz+delo

        return {'dWini':dWini,'dbini':dbini, 'dWin':dWin,'dbin1':dbin1
                ,'dWsec':dWsec,'dbsec':dbsec,'dWout':dWout,'dbout':dbout}

    def forwardGRU(self,data):
        Wa, Ta, ba, Wb, Tb, bb, Wc, Tc, bc=self.We.values()
        numN=self.Size[1] #neuron number
        dims=np.shape(data)
        numS=dims[0] #sample number
        t=dims[1]    #time sequence
        sensor=dims[2]  #three sensor dim

        hiddenL = np.zeros((numS, numN))
        tmp=(numS,t,numN)
        z = np.empty(tmp)
        dz = np.empty(tmp)
        r = np.empty(tmp)
        dr = np.empty(tmp)
        htil = np.empty(tmp)
        dhtil = np.empty(tmp)
        h = np.empty(tmp)

        for i in range(t):
            dataT = data[:, i, :]
            z[:, i, :], dz[:, i, :] = self.activition(np.dot(dataT, Wa) +
hiddenL @ Ta + ba, "sigmoid")
            r[:, i, :], dr[:, i, :] = self.activition(np.dot(dataT , Wb) +
np.dot(hiddenL, Tb) + bb, "sigmoid")
```

```
            mul=np.dot((r[:, i, :] * hiddenL) , Tc)
            htil[:, i, :], dhtil[:, i, :] = self.activition(np.dot(dataT, Wc)
+ mul + bc, "tanh")
            h[:, i, :] = (1 - z[:, i, :]) * hiddenL + z[:, i, :] * htil[:, i,
:]

            hiddenL = h[:, i, :]

        return
hiddenL,{'z':z,'dz':dz,'r':r,'dr':dr,'h':h,'htil':htil,'dhtil':dhtil}

    def backGRU(self,data,cache,dEu):
        _, Ta, _, _, Tb, _, _, Tc, _=self.We.values()
        z=cache['z']
        dz=cache['dz']
        r=cache['r']
        dr=cache['dr']
        h=cache['h']
        htil=cache['htil']
        dhtil=cache['dhtil']
        dWa,dTa,dba,dWb,dTb,dbb,dWc,dTc,dbc=np.zeros((1,9))[0]
        numN=self.Size[1] #neuron number
        dims=np.shape(data)
        numS=dims[0] #sample number
        t=dims[1]    #time sequence
        sensor=dims[2]  #three sensor dim

        for i in reversed(range(t)):
            dataT = data[:, i, :]

            if i > 0:
                hiddenL = h[:, i - 1, :]
            else:
                hiddenL = np.zeros((numS, numN))

            d_z = dEu * (htil[:, i, :] - hiddenL) * dz[:, i, :]
            dh_til = dEu * z[:, i, :] * dhtil[:, i, :]
            d_r = (np.dot(dh_til, Tc.T)) * hiddenL * dr[:, i, :]

            dWa += np.dot(dataT.T, d_z)
            dTa += np.dot(hiddenL.T , d_z)
            dba += np.sum(d_z,axis=0, keepdims=True)

            dWb += np.dot(dataT.T, d_r)
            dTb += np.dot(hiddenL.T , d_r)
            dbb += np.sum(d_r,axis=0, keepdims=True)

            dWc += np.dot(dataT.T , dh_til)
            dTc += np.dot(hiddenL.T, dh_til)
            dbc += np.sum(dh_til,axis=0, keepdims=True)

            d=0
            d +=  (1 - z[:, i, :])*dEu
            d += np.dot(d_z , Ta.T)
            d += np.dot(dh_til  , Tc.T) * (r[:, i, :] + hiddenL *
(np.dot(dr[:, i, :] ,Tb.T)))

        return {"dWa": dWa, "dTa": dTa, "dba": dba,
                "dWb": dWb, "dTb": dTb, "dbb": dbb,
                "dWc": dWc, "dTc": dTc, "dbc": dbc}

    def learn(self,lr,mom,gFL,gMuLP):
        We=self.We
        inLayer=self.inLayer

        MLPparam=self.MLPparam
        MLPm=self.MLPm

        MLPlayer_len=self.MLPlayer_len

        for e in We:
            gUpd = "d"+ e
            inLayer[e] = lr * gFL[gUpd] + mom * inLayer[e]
            We[e] -= inLayer[e]

        for i in range(MLPlayer_len):
            MLPm["W"][i] =  lr * (gMuLP["W"][i]+mom * MLPm["W"][i])
            MLPm["b"][i] = lr * (gMuLP["b"][i] +mom * MLPm["b"][i])
            MLPparam["W"][i] -= MLPm["W"][i]
            MLPparam["b"][i] -= MLPm["b"][i]
            self.w_b_dict = We
            self.input_layer_m = inLayer
            self.mlp_w_b = MLPparam

        self.We=We
        self.inLayer=inLayer
        self.MLPparam=MLPparam
        self.MLPm=MLPm

    def train(self,data,gt,lr,mom,batchS,epoch,modelType):
        np.random.seed(666)
        lostTrain = []
        lostVal = []
        accTrain = []
        accVal = []

        sampleNum=data.shape[0]
        validation= sampleNum//10
        randomData = np.random.permutation(sampleNum)

        valData=data[randomData][:validation]
        valGt=gt[randomData][:validation]
        trainData=data[randomData][validation:]
        trainGt=gt[randomData][validation:]

        iteration=int(sampleNum/batchS)

        for i in range(epoch):
            s=0
            e=batchS
            rand = np.random.permutation(trainData.shape[0])
            trainData = trainData[rand]
            trainGt = trainGt[rand]
            for j in range(iteration):

                dataBatch = trainData[s:e]
                gtBatch = trainGt[s:e]

                choice,out,d,hidden,dHidden,cache =
self.forward(dataBatch,modelType=modelType)
                dE = choice
                dE[gtBatch == 1] -= 1
                dE /= batchS

                gRNN, gMLP = self.backward(dataBatch, out, d, dE,
modelType,hidden, dHidden, cache )
                self.learn(lr, mom, gRNN, gMLP)
                s = e
                e=e+ batchS

            trainacc, trainP, trainconf = self.choice(trainData,trainGt)
            trainloss = self.CE(trainGt, trainP)

            valAcc, valP, valConf = self.choice(valData,valGt)
            valloss = self.CE(valGt, valP)

            lostTrain.append(trainloss)
            lostVal.append(valloss)
            accTrain.append(trainacc)
            accVal.append(valAcc)
            print('\nTraining Loss: ', trainloss, 'Validation Loss: ',
valloss, 'Training Accuracy: ', 'Validation Accuracy: ',trainacc, 'Epoch:
',i+1, 'out of', epoch)

            if (i > 15) and ( modelType!='gru'):
                avlost = lostVal[-9:-1] #check last 8 arbitrary choice
                avlost = sum(avlost) / len(avlost)
                limit = 0.005
                if (avlost - limit) < valloss < (avlost + limit):
                    print("\nTraining ends due to convergence.")
                    return {"lostTrain": lostTrain, "lostVal": lostVal,
                            "accTrain": accTrain, "accVal":
accVal,'trainconf':trainconf}
        return {"lostTrain": lostTrain, "lostVal": lostVal,
                            "accTrain": accTrain, "accVal": accVal,
'trainconf':trainconf}

    def choice(self,data,gt=None):
        modelType=self.modelType0
        p,_,_,_,_,_=self.forward(data,modelType=modelType)
        predictions=p.argmax(axis=1)
        gt=gt.argmax(axis=1)

        accuracy=0
        confidence=np.zeros((6,6))
        for i in range(data.shape[0]):
            confidence[gt[i]][predictions[i]]+=1
            if gt[i]==predictions[i]:
                accuracy+=1
        accuracy/=len(data)
        return accuracy, p, confidence

    def CE(self,gt,choi):
        return np.sum(-gt*np.log(choi))/gt.shape[0]

    def activition(self,data,activation):
        if activation=='softmax':
            Z=np.exp(data)/np.sum(np.exp(data),axis=1,keepdims=True)
            d=None
        elif activation=='sigmoid':
            Z=1/(1+np.exp(-data))
            d=Z-Z**2
        elif activation=='relu':
            #Z=np.max(0.0,data)
            Z = data * (data > 0)
            d=1*(data>0)
        elif activation=='tanh':
            Z=np.tanh(data)
            d=1- Z**2
        return Z,d
############End##################
import sys

question =sys.argv[1]
def sereftaha_kiremitci_21903711_miniproject(question):
```

```python
    print('Question ',question)
    if question == '1' :
        Q1()
    elif question == '2' :
        q2()
    elif question == '3' :
        q3()

sereftaha_kiremitci_21903711_miniproject(question)
```