Mustafa Cankan Balcı
22101761
22 December 2024

EEE 443- 543 Neural Network – Mini Project

Detailed Information

Mini project aims to design different kinds of neural networks for different applications and observe their results. Mini project has 3 parts. In part 1, sparse autoencoder is designed to obtain hidden feature extraction. Moreover, natural language structure is designed for predicting the next word according to given input. Finally, sequential neural network models such as RNN, LSTM and GRU designed to predict the movement.

Part 1

In the part of mini project, sparse autoencoder is designed for unsupervised feature extraction. The neural network structure is figure 1. Autoencoder is constructed on 2 parts: encoder and decoder. Also, each section weights are tied. As a result, the calculation and weights initialized depending on tied weights criteria. In the encoder and decoder part of autoencoder, the sigmoid is used as activation function.

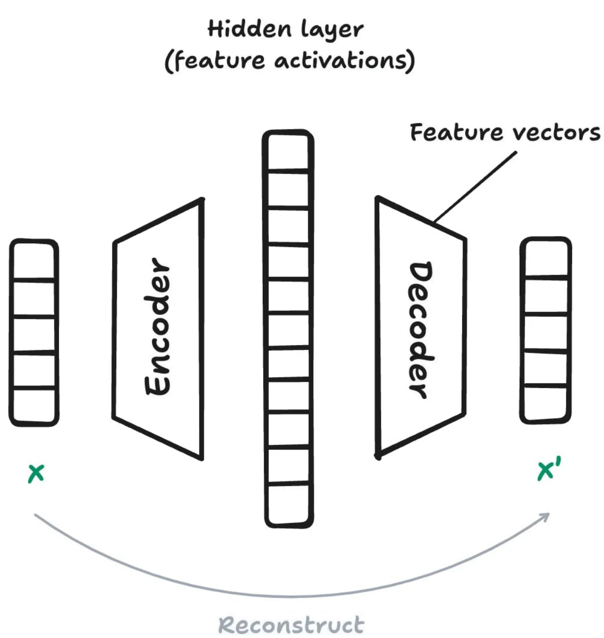


Fig. 1: Autoencoder Structure

Section A

In the section A, the data set of part 1 is preprocessed. RGB images in the data set turned into grayscale by using the luminosity model. For normalizing data, mean pixel intensity of each image subtracted from itself and clipped +/- 3 standard deviations, which is measured from all pixels in the data. To prevent the activation function from saturating, scaled the data range of +/- 3 standard deviations to interval [0.1,0.9].

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

For the visualition of the preprocessing process, randomly 200 images selected in the dataset. Normalized version of RGB images are gray scale versions of them. Edges in the can be easily detected in the grayscale normalized version, so autoencoder can easily find the hidden features.

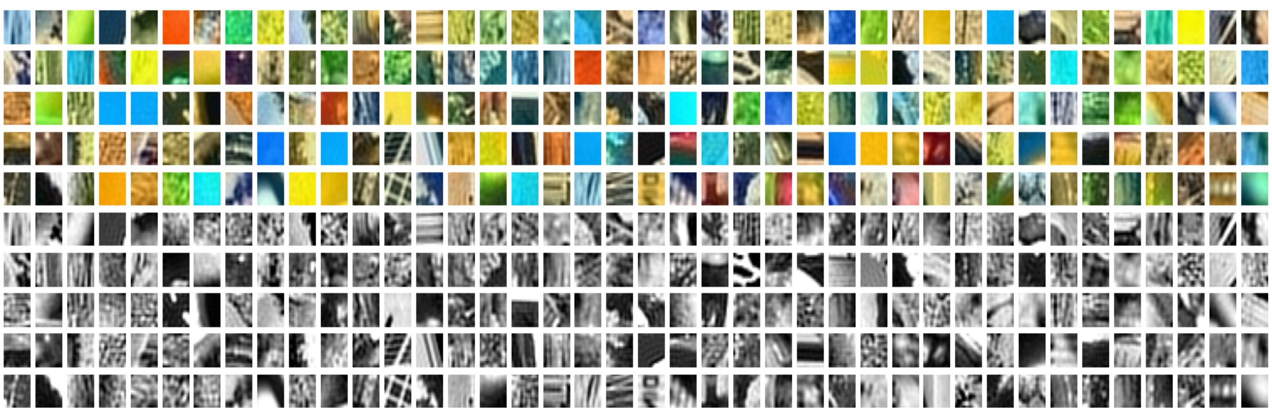


Fig. 2: Section A Output

Section B

First of all, weights and biases of autoencoder parts are initialized randomly from the uniform of $[-w_o, w_o]$. w_o is determined by formula, where $L_{pre,post}$ are neurons on either side of the connection weights. The corresponding formula is shown below.

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

Afterwards, cost function for the network is written for calculating the cost and its partial derivatives. The cost function for part A consists of 3 parts. The first term of cost is average square error between desired output and predicted output of autoencoder. The second term enforces Tykhonov regularization (L2 regularization) on the connection weights. λ. The level of sparsity in determined by the KL divergence between with mean ρ and with mean $\widehat{\rho_j}$.

$$J = \frac{1}{2N}\sum_{i=1}^{N} |\, d_i - y_i|^2 + \frac{\lambda}{2}(|W_1|^2 + |W_2|^2) + \beta \sum_{j=1}^{L_{\text{hidden}}} \left(\rho \log \frac{\rho}{\widehat{\rho_j}} + (1-\rho)\log \frac{1-\rho}{1-\widehat{\rho_j}}\right)$$

Forward pass equations of the autoencoder are:

$$h = \sigma(\text{data} \cdot W_1 + b_1)$$

$$y = \sigma(h \cdot W_2 + b_2)$$

Moreover, gradients respect to cost function is calculated for updating paratemers. Their gradients respect to cost function are :

$$\frac{\partial J_{\text{reconstruction}}}{\partial y} = \frac{1}{N}(y - d)$$

$$\delta_2 = \frac{1}{N}(y - d) \odot y(1 - y)$$

$$\frac{\partial J}{\partial W_2} = h^T \cdot \delta_2 + \lambda W_2$$

$$\frac{\partial J}{\partial b_2} = \sum_{i=1}^{N} \delta_2$$

$$\frac{\partial J_{\text{KL}}}{\partial \hat{\rho}} = \beta\left(-\frac{\rho}{\hat{\rho}} + \frac{1-\rho}{1-\hat{\rho}}\right)$$

$$\delta_1 = \left(\delta_2 \cdot W_2^T + \frac{\partial J_{\text{KL}}}{\partial \hat{\rho}}\right) \odot h \odot (1 - h)$$

$$\frac{\partial J}{\partial W_1} = d^T \cdot \delta_1 + \lambda W_1$$

$$\frac{\partial J}{\partial b_1} = \sum_{i=1}^{N} \delta_1$$

According to gradient calculations and cost function formula, the aeCost function is designed to find the cost and partial derivatives.

```
def aeCost(We,data,params):
    lambda_reg = params["lambda"]
    rho = params["rho"]
    beta = params["beta"]
    N = data.shape[0]
    W1,b1,W2,b2= We
    outputs = forward_prob(We,data)
    y = outputs["Output of Autoencoder"]
    h = outputs["Hidden Output"]
    d = data
    # Calculate the cost
    rho_b = h.mean(axis=0, keepdims=True)  # 1 x Lhid
    b = np.sum(W1 ** 2)
    c = np.sum(W2 ** 2)
    KL = rho * np.log(rho/rho_b) + (1 - rho) * np.log((1 - rho)/(1 - rho_b))
    KL = beta * KL.sum()
    J = 0.5/N * (np.linalg.norm(d - y, axis=1) ** 2).sum() + 0.5 * lambda_reg * (b +
c) + KL
     # Encoder Backprob
    dy = (y-d)/N
    dy1 = y*(1-y)*dy
    dW2 = np.dot(h.T,dy1) + lambda_reg*W2
    db2 = dy1.sum(axis=0,keepdims=True)
    # Without KL Divergence Decoder
    dpj = beta*(-rho/rho_b + (1-rho)/(1-rho_b))/N
    dh = np.dot(dy1,W2.T) + dpj
    dh1 = h*(1-h)*dh
    dW1 = np.dot(d.T,dh1) + lambda_reg*W1
    db1 = dh1.sum(axis=0,keepdims=True)
    dW2 = (dW1.T + dW2)
    dW1 = dW2.T
    dWe = [dW1, db1, dW2, db2]
    return J,dWe
```

For updating parameters in autoencoder structure, momentum with mini batch stochastic gradient descent is used for updating parameters in the solver function.

```
def solver(grads,We,learning_rate,momentum,alpha):
    W1,b1,W2,b2 = We
    dW1,db1,dW2,db2 =  grads

    momentum["W1"] = alpha *momentum["W1"] - learning_rate*dW1
    W1 = W1 + momentum["W1"]
    momentum["b1"] = alpha *momentum["b1"] - learning_rate*db1
    b1 = b1 + momentum["b1"]
    momentum["W2"] = alpha *momentum["W2"] - learning_rate*dW2
    W2 = W2 + momentum["W2"]
```

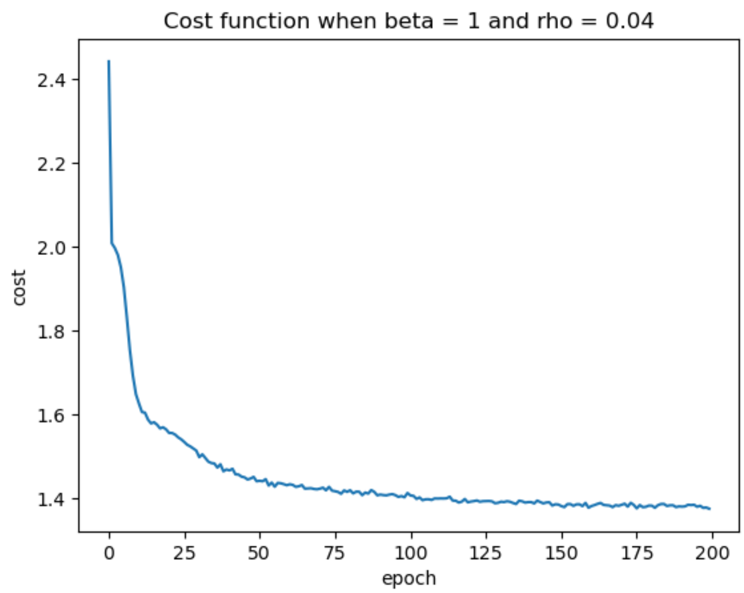
```
    momentum["b2"] = alpha *momentum["b2"] - learning_rate*db2
    b2 = b2 + momentum["b2"]

    return [W1,b1,W2,b2],momentum
```

In the training process, written functions are used to train autoencoder according to training parameters. In start of each epoch, training set is shuffled to overcome the memorization in the network. According to observation section B, table 1 represents the optimized values.

Learning Rate (η)	0.01
Momentum Rate (α)	0.85
Epoch	200
Beta	1
Rho	0.04
Mini Batch Size	32
Number of Hidden Neurons	64

Table 1: Part A Parameters



Graph 1: Cost Function Graph

Moreover, the performance of autoencoder is criticized how well the hidden features. I also looked the reconstruction of images. Figure 3 shows the reconstruction of figure 4. The reconstrued image is noise version of it since the model is based on sparsity. As a result, the autoencoder has good reconstruction capability.

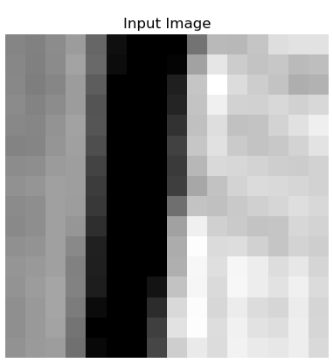


Fig. 3: Input Image

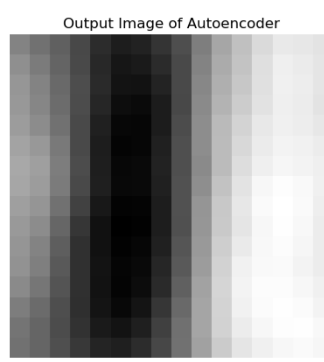


Fig. 4: Output Image

Section C

Afterwards, the first layer of connection weights showed in the separate image in the hidden layer. Figure 5 represents the hidden layer features. The hidden features represent pattern of natural images. It does not show directly the image, but patterns and edges of the image is shown in hidden feature representation. The hidden features retain meaningful structures like edges, gradients and localized blobs. The are different types of features with range of orientations such as horizontal, vertical, and diagonal. Generally, the noise of hidden filters is low noise.

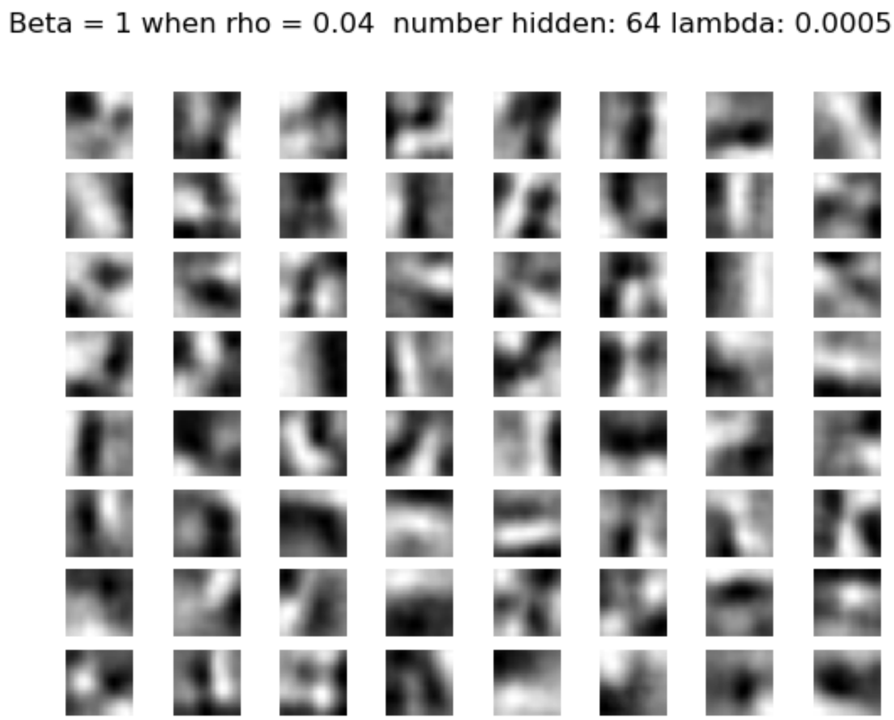


Fig. 5: Hidden Feature Section C

Section D

The autoencoder model was trained using different hidden layer sizes such as 10, 64, and 100 neurons, and various λ values such as 0, 0.001, and 0.0005. Also, β and ρ are kept constant for this part. A total of 9 sets of graphs depicting the hidden features were generated to observe the effects of λ and the number of hidden neurons on feature extraction.

Based on the observations, λ significantly influences the smoothness of the features. Higher λ values result in increased smoothness, whereas the absence of L2 regularization ($\lambda = 0$) leads to features that capture finer details without penalty, making edge detection challenging.

Increasing the number of hidden neurons adds complexity to the model. However, in the context of feature extraction, redundancy is observed as the number of hidden neurons increases. For instance, with 10 hidden neurons, each feature is unique and distinct, whereas with 64 and 100 neurons, certain features are repeated among the hidden units. As a result, there exists a trade-off between model complexity and feature uniqueness. While a smaller number of hidden neurons ensures distinct features, larger hidden layers introduce redundancy in feature extraction.

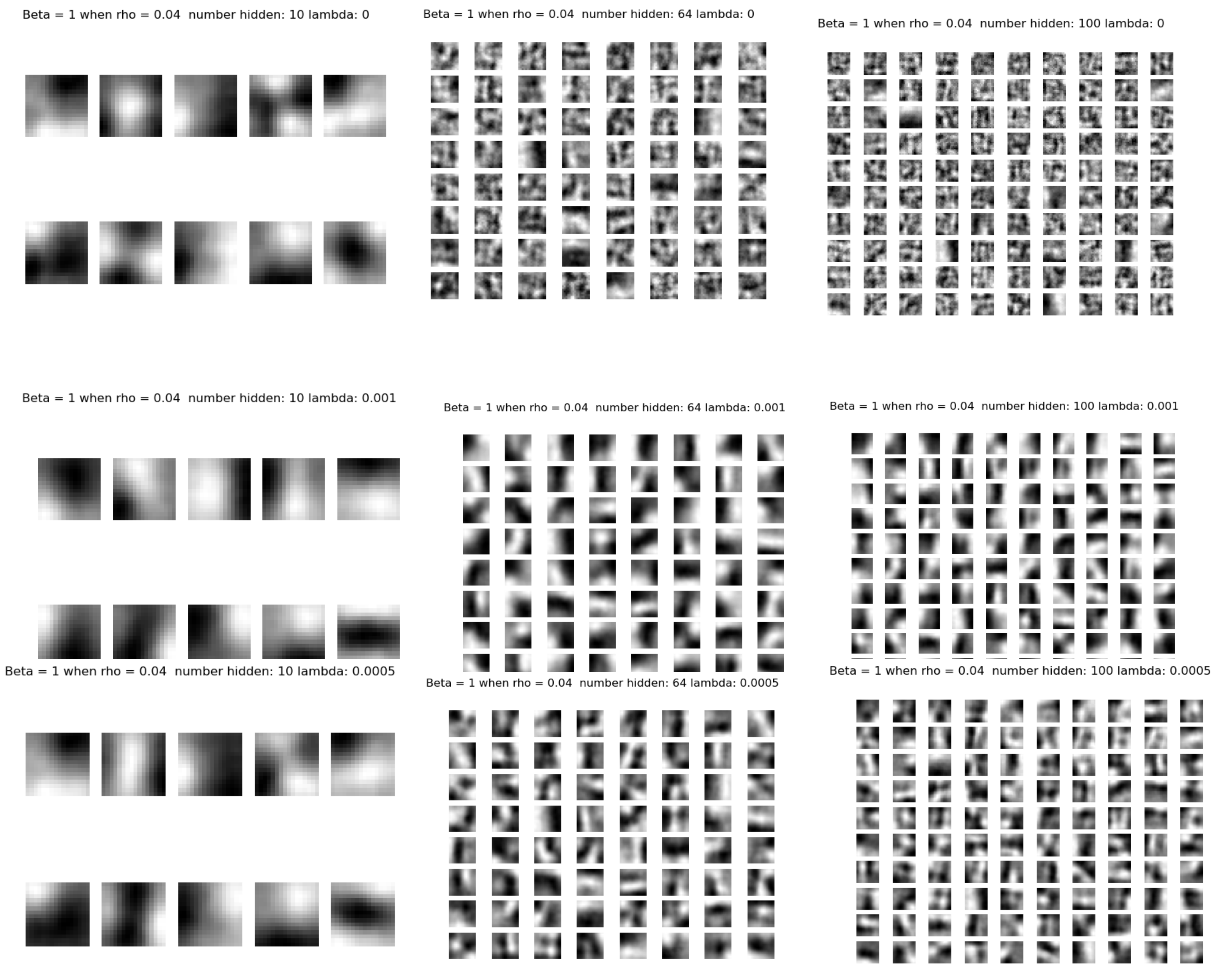


Fig. 6: Output Features Section D Part 1

Part 2

In the part 2 of mini project, neural network structure is designed for natural language processing. The task is to predict the fourth word in sequence given the preceding trigram. The data set files contain test, train and validation data set. The output layer is the SoftMax output. It is score of each predicted word. In the network 250 words scores is generated at the output layer. In the optimization of NLP network, momentum with SGD is chosen.

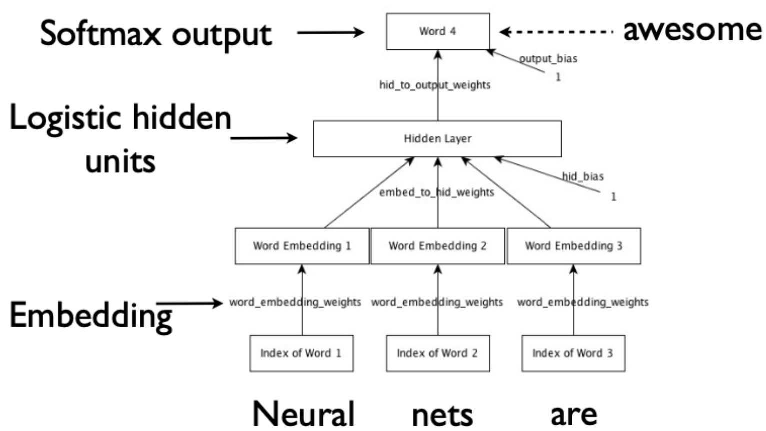


Fig. 7: Part 2 Neural Network Structure

The forward pass of the neural network is shown below:

$$\text{embedding1} = W_{\text{embedded\_weights}}[\text{list\_embedded}[:,0] - 1]$$

$$\text{embedding2} = W_{\text{embedded\_weights}}[\text{list\_embedded}[:, 1] - 1]$$

$$\text{embedding3} = W_{\text{embedded\_weights}}[\text{list\_embedded}[:,2] - 1]$$

$$\text{embeddings} = [\text{embedding1,embedding2,embedding3}]$$

$$z_h = \text{embeddings} \cdot W_{ih} + b_{ih}$$

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

$$z_o = h \cdot W_{ho} + b_{ho}$$

$$y = \frac{e^{z_{o,i}}}{\sum_{j=1}^{c} e^{z_{o,j}}}$$

The cost function of part 2 is cross entropy loss function.

$$\mathcal{L} = -\sum_{i=1}^{C} y_{\text{true},i} \cdot \log(y_{\text{pred},i}) = -log(y_{\text{pred,i}*})$$

Gradients Output Layer :

$$\frac{\partial \mathcal{L}}{\partial z_o} = y_{\text{pred}} - y_{\text{true}}$$

$$\frac{\partial \mathcal{L}}{\partial W_{\text{ho}}} = \frac{1}{m} h^T \cdot \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\frac{\partial \mathcal{L}}{\partial b_{\text{ho}}} = \frac{1}{m} \sum \frac{\partial \mathcal{L}}{\partial z_o}$$

Gradients for Hidden Layer:

$$\frac{\partial \mathcal{L}}{\partial h} = \frac{\partial \mathcal{L}}{\partial z_o} \cdot W_{\text{ho}}^T$$

$$\frac{\partial \mathcal{L}}{\partial z_h} = \frac{\partial \mathcal{L}}{\partial h} \cdot h \cdot (1 - h)$$

$$\frac{\partial \mathcal{L}}{\partial W_{\text{ih}}} = \frac{1}{m} \text{embeddings}^T \cdot \frac{\partial \mathcal{L}}{\partial z_h}$$

$$\frac{\partial \mathcal{L}}{\partial b_{\text{ih}}} = \frac{1}{m} \sum \frac{\partial \mathcal{L}}{\partial z_h}$$

Gradients for Embedding Layer:

$$\frac{\partial \mathcal{L}}{\partial \text{embeddings}} = \frac{1}{m} \frac{\partial \mathcal{L}}{\partial z_h} \cdot W_{\text{ih}}^T$$

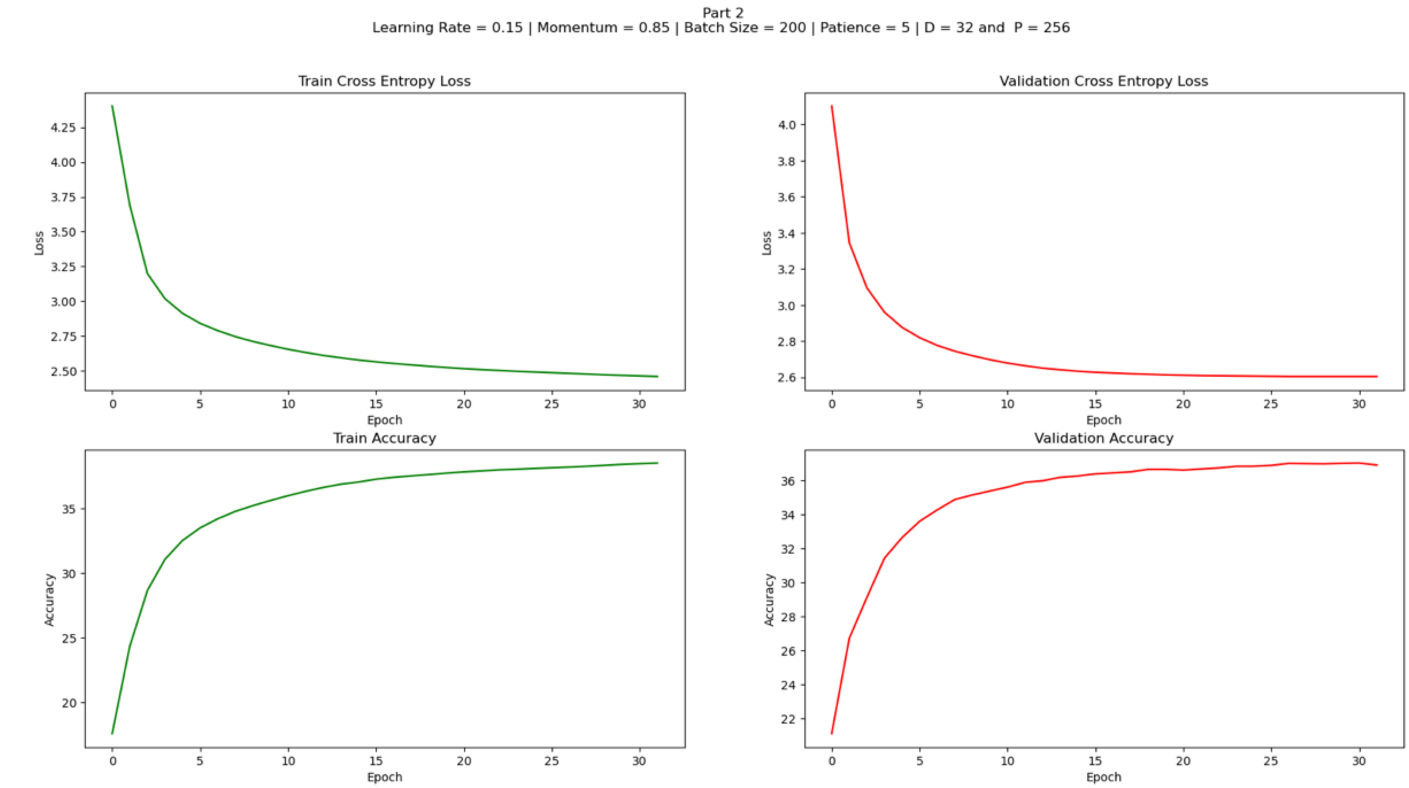
$$\frac{\partial \mathcal{L}}{\partial W_{\text{embedded\textbackslash weights}}} [xi, j] += \frac{\partial \mathcal{L}}{\partial \text{embeddings}} i, j$$

Section A

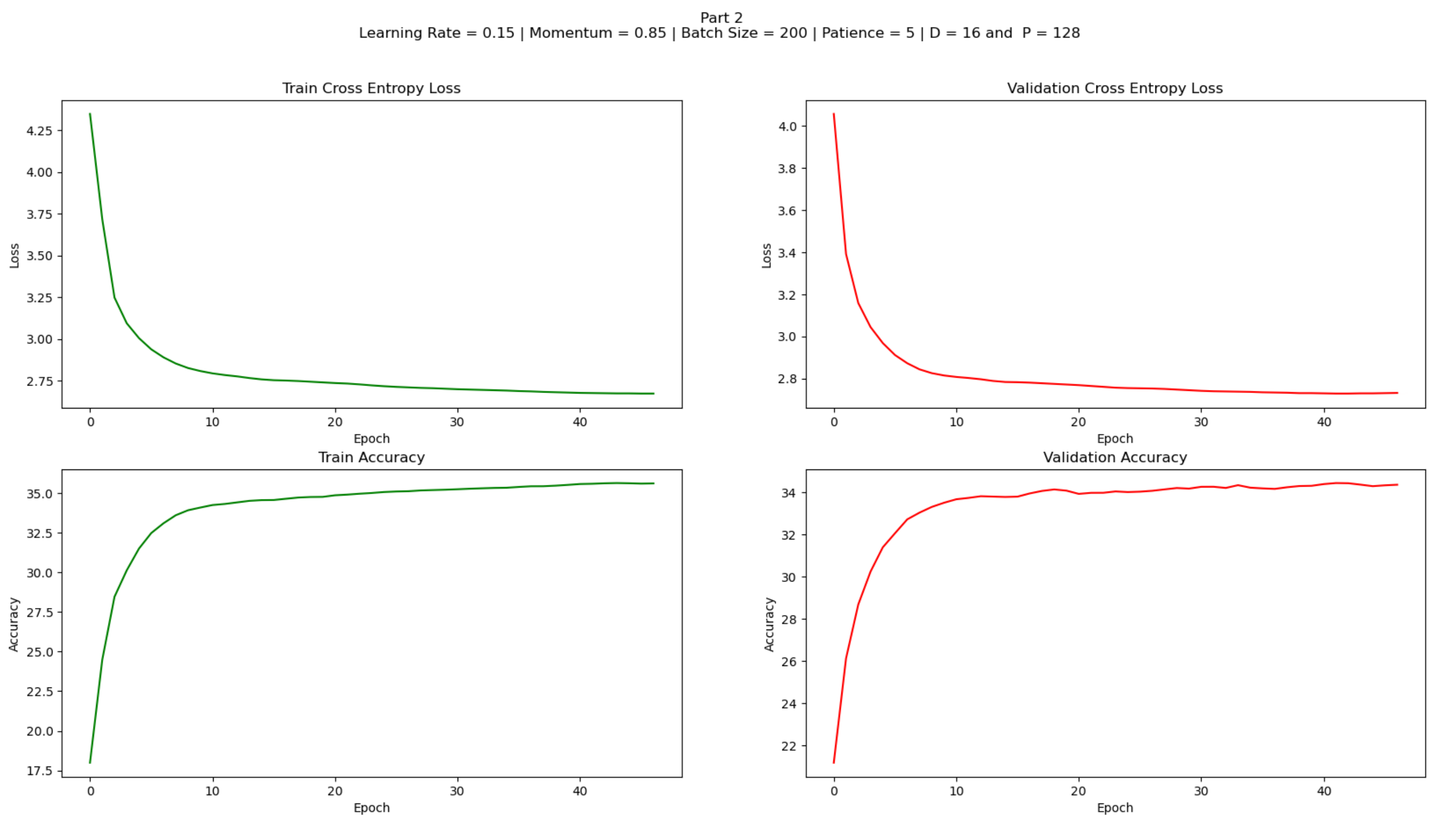
The parameters used in the training process of neural network is given in table 2. Begin of each epoch, train data set is shuffled to overcome the memorization issue. These parameters expect patience are same as the given values in the mini project report. Patience is declared as 5 for part B.

Learning Rate (η)	0.15
Momentum Rate (α)	0.85
Epoch	50
Patience	5
Mini Batch Size	200

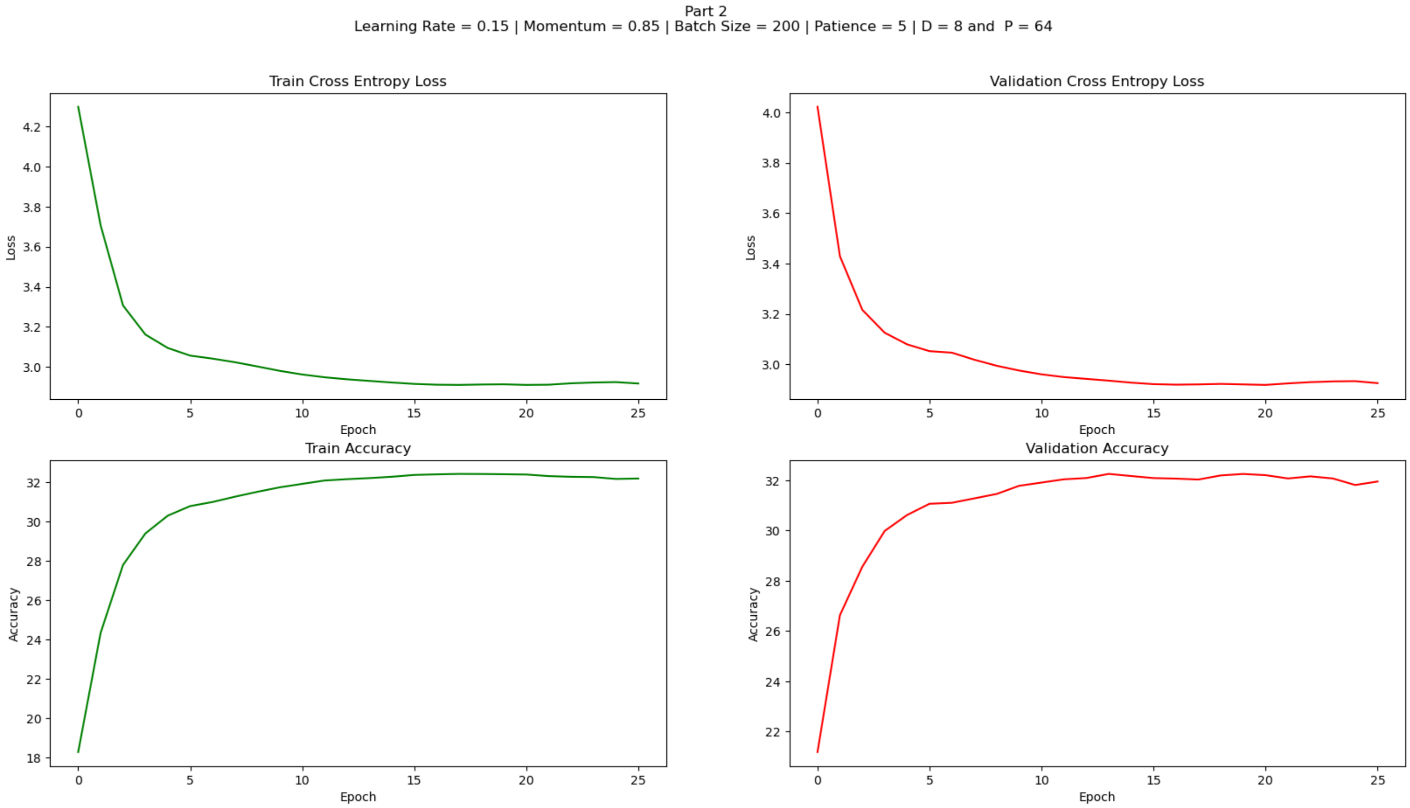
Table 2: Part B Parameters



Graph 2: Graphs for Training Parameters when D= 32 P =256



Graph 3: Graphs for Training Parameters when D= 16 P =128



Graph 4: Graphs for Training Parameters when D=8 P =64

Table 3 represents the validation accuracy in their last epoch. Graph 2, 3, and 4 shows the calculated values in the graphs. According to observations in graphs, when the D and P values decrease, validation accuracy decreases. Also, training and validation accuracy is increasing until one point, and model stops with early stopping. Consequently, models' validation accuracy decreases since model complexity decreases.

(D, P)	Validation Accuracy
(32,256)	36.91 %
(16,128)	34.36 %
(8,64)	31.95 %

Table 3: Validation Accuracy for Different (D,P values)

Section B

Furthermore, 5 sample trigram are selected randomly from test set of part B. D and P values selected as 32 and 256 respectively in this section. Afterwards, top 10 predictions of 5 sample trigram are shown in the table. If the word is predicted correctly, it has higher probability than other predictions. If the predicted word with highest probability is not same as expected word in test set, it was predicted meaningfully. For example, 2nd sample 4th word is ".", but NLP predicted "not" which is grammatically correct. "The money is" does not follow up with ".".

There is some explanation about is expected like “not” or been in grammatically. In all 5 cases, 10th prediction is less meaningful than 1st prediction. However, some parts are not correct in the dataset. For solving this issue, data number can be increased or relabeling can be changed with high accuracy.

1st Sample	in	his	life	**.**
2nd Sample	the	money	is	**.**
3rd Sample	she	just	does	**nt**
4th Sample	in	the	right	**place**
5th Sample	still	going	on	**?**

Table 4: 5 Randomly Selected Trigram and Expected Word

.	,	public	week	like	were	should	war	even	war
0.7025	0.1758	0.0496	0.0060	0.0050	0.0036	0.0034	0.0025	0.0024	0.0023

Table 5: Top 10 Prediction of 1st Sample

not	been	west	use	to	only	well	take	still	political
0.1527	0.1278	0.1166	0.0977	0.0670	0.0448	0.0253	0.0227	0.0178	0.0162

Table 6: Top 10 Prediction of 2nd Sample

nt	.	not	,	have	more	out	year	new	school
0.8619	0.050	0.0168	0.0160	0.0139	0.0068	0.0051	0.0024	0.0020	0.0016

Table 7: Top 10 Prediction of 3rd Sample

place	too	been	,	year	i	world	team	on	it
0.5990	0.1662	0.0841	0.0187	0.0175	0.0171	0.01515	0.0057	0.0056	0.0029

Table 8: Top 10 Prediction of 4th Sample

.	?	for	,	business	case	all	little	including	ms.
0.7516	0.0685	0.0300	0.0225	0.0221	0.0164	0.0134	0.0091	0.0091	0.0057

Table 9: Top 10 Prediction of 5th Sample

Part 3

In the part 3, human activities tried to be classified based on movement signals captured from 3 sensors operating simultaneously. The corresponding number of the activity is shown in table 11. Each time series has a length of 150 units. The training set contains 3000 samples, and the test set includes 600 samples. Different sequential neural network architectures are designed to predict the human activity.

downstairs	1
jogging	2
Sitting	3
Standing	4
Upstairs	5
Walking	6

Table 10: Human Activities Classification

Section A

In the section A, recurrent neural network (RNN) is designed. The difference between RNN and feedforward neural network is that RNN has the feedback loop, which allows memorization in the network. RNN is used for sequential processing [1]. The structure of RNN is shown in figure 8.

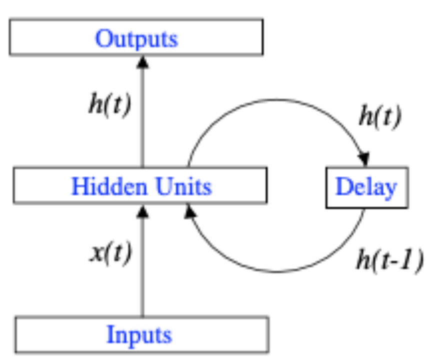


Fig. 8: RNN Structure

The activation functions of hidden and output are tanh(x) and SoftMax respectively.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The forward pass of RNN is shown below:

$$h_t = \tanh(x_t W_{xh} + h_{t-1} W_{hh} + b_x)$$

$$y_t = \text{softmax}(h_t W_{ho} + b_o)$$

The cost funciton in RNN is cross entropy loss, since we are using multiple class classficaiton.

$$\mathcal{L}(y_{\text{true}}, y_{\text{pred}}) = -\sum_{i=1}^{C} y_{\text{true},i} \cdot \log(y_{\text{pred},i})$$

Normally, RNN cannot apply the backpropagation directly. Feedback loop in the network restricts the backpropagation application since network structure becomes time dependent. As a result, RNN structure unfolds overtime period. By this way, RNN behaves like a feedforward neural network when it unfolds. This process is called backpropagation through time. In each section, many-to-one structure is used for BPTT.

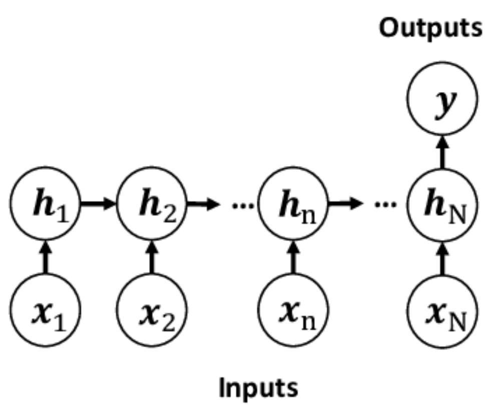


Fig. 9: BPTT Structure [2]

The gradients are calculated for updating parameters with using BPTT.

$$\frac{\partial \mathcal{L}}{\partial z_o} = y_{\text{pred}} - y_{\text{true}}$$

$$\frac{\partial \mathcal{L}}{\partial W_{ho}} = h_t^T \cdot \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\frac{\partial \mathcal{L}}{\partial b_o} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\delta_t = \frac{\partial \mathcal{L}}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial z_o} \cdot W_{ho}^T + \delta_{t+1} \cdot W_{hh}^T$$

$$\frac{\partial \mathcal{L}}{\partial h_t} = \delta_t \cdot (1 - h_t^2)$$

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t=1}^{T} X_t^T \cdot \delta_t$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=2}^{T} h_{t-1}^T \cdot \delta_t$$

$$\frac{\partial \mathcal{L}}{\partial b_x} = \sum_{t=1}^{T} \delta_t$$

$$\frac{\partial \mathcal{L}}{\partial b_o} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\delta_t = \frac{\partial \mathcal{L}}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial z_o} \cdot W_{ho}^T + \delta_{t+1} \cdot W_{hh}^T$$

$$\frac{\partial \mathcal{L}}{\partial h_t} = \delta_t \cdot (1 - h_t^2)$$

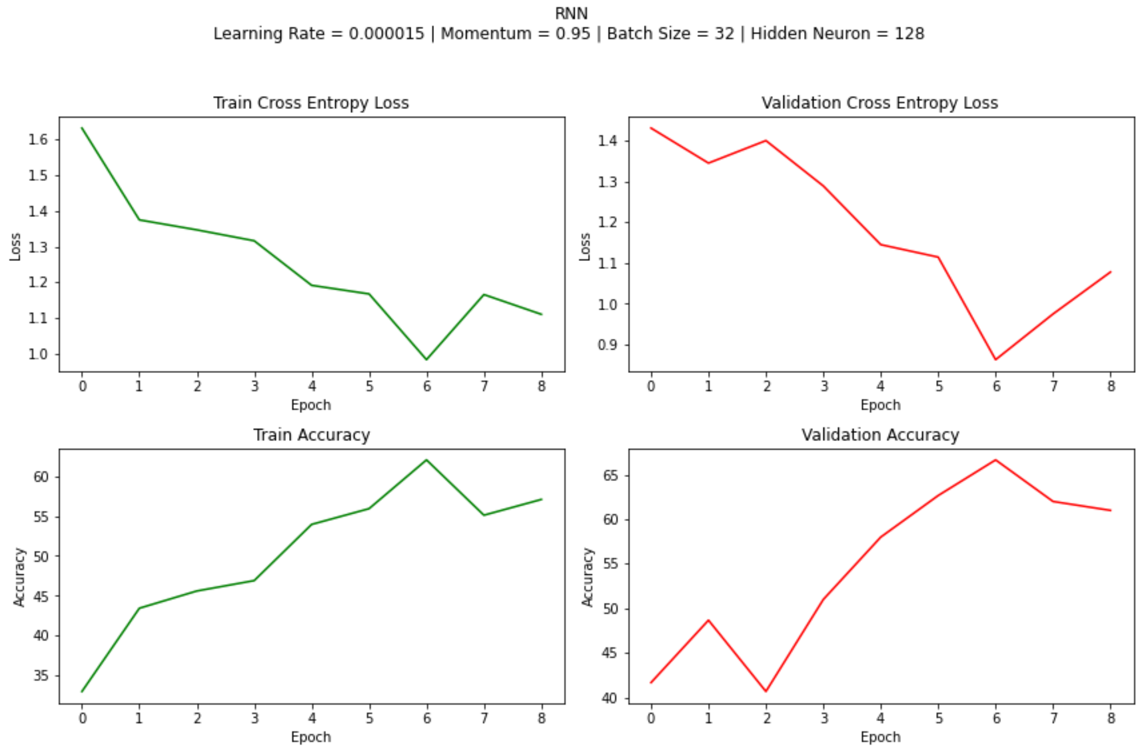
In the optimization of RNN training, momentum with SGD is selected as optimizer. Start of each epoch training set is shuffled and separated 10 % for validation set. The parameters are adjusted to obtain high accuracy in validation and low cost at training and validation. Table 11 shows the parameters used in RNN training process. Learning rate is kept small since gradients are exploit in large time interval.

Learning Rate	0.000015
Momentum Rate	0.95
Epoch	50
Patience	3
Batch Size	32
Number of Hidden Neurons	128

Table 11: Training Parameters RNN

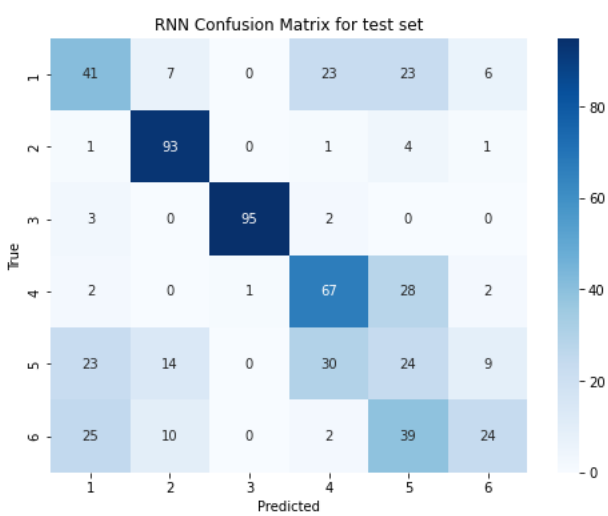
Train Cost	1.111
Validation Cost	1.078
Train Accuracy	61.70 %
Validation Accuracy	61.00 %
Test Accuracy	57.33 %

Table 12: Obtained Parameters RNN

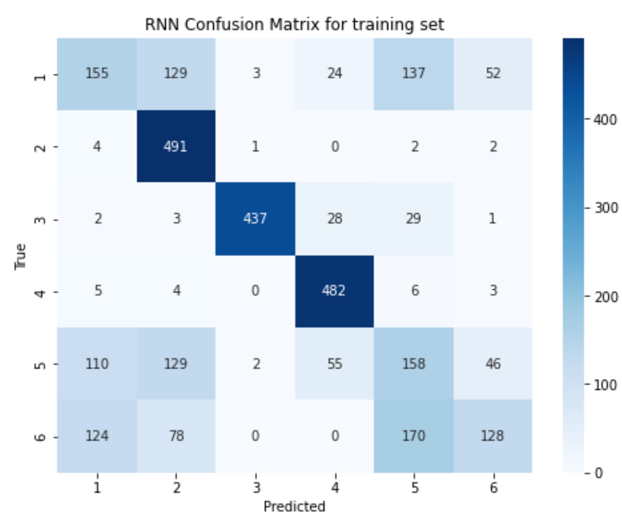


Graph 5: Graphs for Training Parameters for RNN

The RNN's training and validation curves show a promising but flawed learning process. While validation loss has a similar trend with sporadic spikes, indicating some overfitting or noise in the data, training loss constantly declines, showing effective model modification. Both training and validation accuracy metrics demonstrate consistent gains, although validation accuracy varies noticeably, indicating difficulties with generalization. The spikes occur in loss graphs because of exploding gradients. Overall, the model shows promise for learning, but it has to be improved for reliable operation.



Graph 6: Confusion Matrix Test for RNN



Graph 7: Confusion Matrix Train for RNN

According to confusion matrix of test and train set, RNN confuses the predict human activity expect class 2 and 3. In the training set, diagonal values are prominent, indicating that model preforms well on the training data. A notable number of samples from class 1 misclassified as class 5 or 6 in both cases.

Section B

In section B, long term short memory (LSTM) model is implemented. Figure 10 represents the LSTM schematics. LSTMs have a "memory cell" that retains information over time. There are 4 types of gates: input gate, forget gate a candidate cell state gate, output gate. Forget gate decides what information to discard from the cell state. Input gate determines what new information should be added to the cell state. Finally, output gate controls the output based on the cell state and decides what to send to the next layer or timestep. In the neural network structure of section B, LSTM output is connected to SoftMax layer for multiclass classification

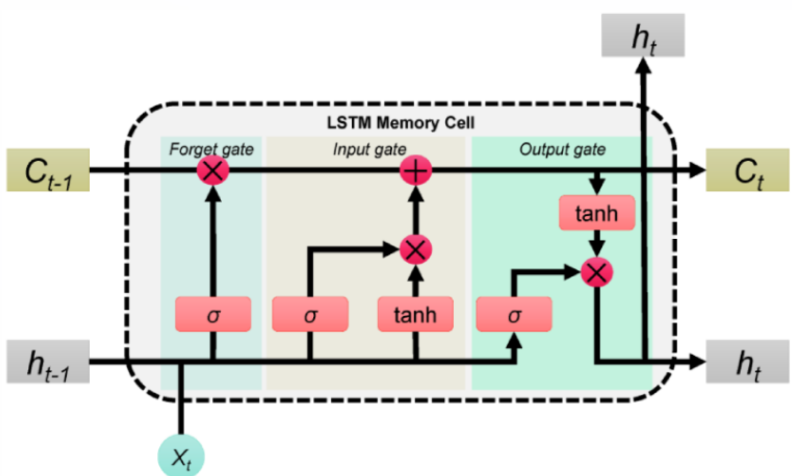


Fig. 10: LSTM Structure [3]

The LSTM consists of different type of gates. The forward propagation formulas of gates are :

$$f_t = \sigma\left(x_t W_f + h_{t-1} U_f + b_f\right)$$

$$i_t = \sigma(x_t W_i + h_{t-1} U_i + b_i)$$

$$\tilde{c}t = \tanh(x_t W_c + ht - 1U_c + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c_t}$$

$$o_t = \sigma(x_t W_o + h_{t-1} U_o + b_o)$$

$$h_t = o_t \odot \tanh(c_t)$$

$$y_{\text{output}} = h_T \cdot W_{\text{dense}} + b_{\text{dense}}$$

Gradients for each gate parameters are calculated for optimization process. Additionally, gradient clipping is applied to overcome the exploding gradient issue.

```
for grad in grads.values():
            np.clip(grad, -5, 5, out=grad)
```

Gradient Output Layer of LSTM:

$$\frac{\partial \mathcal{L}}{\partial z_o} = y_{\text{pred}} - y_{\text{true}}$$

$$\frac{\partial \mathcal{L}}{\partial W_{\text{hd}}} = h_T^T \cdot \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\frac{\partial \mathcal{L}}{\partial b_{\text{hd}}} = \sum \frac{\partial \mathcal{L}}{\partial z_o}$$

Gradient Forget Gate of LSTM:

$$\frac{\partial \mathcal{L}}{\partial f_t} = \frac{\partial \mathcal{L}}{\partial c_t} \cdot c_{t-1}$$

$$\frac{\partial \mathcal{L}}{\partial z_f} = \frac{\partial \mathcal{L}}{\partial f_t} \cdot f_t \cdot (1 - f_t)$$

$$\frac{\partial \mathcal{L}}{\partial W_f} += [h_{t-1}, x_t]^T \cdot \frac{\partial \mathcal{L}}{\partial z_f}$$

$$\frac{\partial \mathcal{L}}{\partial b_f} += \sum \frac{\partial \mathcal{L}}{\partial z_f}$$

Gradient Input Gate of LSTM:

$$\frac{\partial \mathcal{L}}{\partial i_t} = \frac{\partial \mathcal{L}}{\partial c_t} \cdot \tilde{c}t$$

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\partial \mathcal{L}}{\partial i_t} \cdot i_t \cdot (1 - i_t)$$

$$\frac{\partial \mathcal{L}}{\partial W_i} += [ht - 1, x_t]^T \cdot \frac{\partial \mathcal{L}}{\partial z_i}$$

$$\frac{\partial \mathcal{L}}{\partial b_i} += \sum \frac{\partial \mathcal{L}}{\partial z_i}$$

Gradient Candidate Gate of LSTM:

$$\frac{\partial \mathcal{L}}{\partial \tilde{c_t}} = \frac{\partial \mathcal{L}}{\partial c_t} \cdot i_t$$

$$\frac{\partial \mathcal{L}}{\partial z_c} = \frac{\partial \mathcal{L}}{\partial \tilde{c}_t} \cdot (1 - \tilde{c}t^2)$$

$$\frac{\partial \mathcal{L}}{\partial W_c} += [ht - 1, x_t]^T \cdot \frac{\partial \mathcal{L}}{\partial z_c}$$

$$\frac{\partial \mathcal{L}}{\partial b_c} += \sum \frac{\partial \mathcal{L}}{\partial z_c}$$

Gradient Output Gate of LSTM:

$$\frac{\partial \mathcal{L}}{\partial o_t} = \frac{\partial \mathcal{L}}{\partial h_t} \cdot \tanh(c_t)$$

$$\frac{\partial \mathcal{L}}{\partial z_o} = \frac{\partial \mathcal{L}}{\partial o_t} \cdot o_t \cdot (1 - o_t)$$

$$\frac{\partial \mathcal{L}}{\partial W_o} += [h_{t-1}, x_t]^T \cdot \frac{\partial \mathcal{L}}{\partial z_o}$$

$$\frac{\partial \mathcal{L}}{\partial b_o} += \sum \frac{\partial \mathcal{L}}{\partial z_o}$$

Gradient for Cell State:

$$\frac{\partial \mathcal{L}}{\partial c_t} = \frac{\partial \mathcal{L}}{\partial h_t} \cdot o_t \cdot (1 - \tanh^2(c_t))$$

Gradient for Previous timestep:

$$\frac{\partial \mathcal{L}}{\partial h_{t-1}} = \left[\frac{\partial \mathcal{L}}{\partial f_t} W_f + \frac{\partial \mathcal{L}}{\partial i_t} W_i + \frac{\partial \mathcal{L}}{\partial \tilde{c}t} W_c + \frac{\partial \mathcal{L}}{\partial o_t} W_o\right] h_{t-1}$$

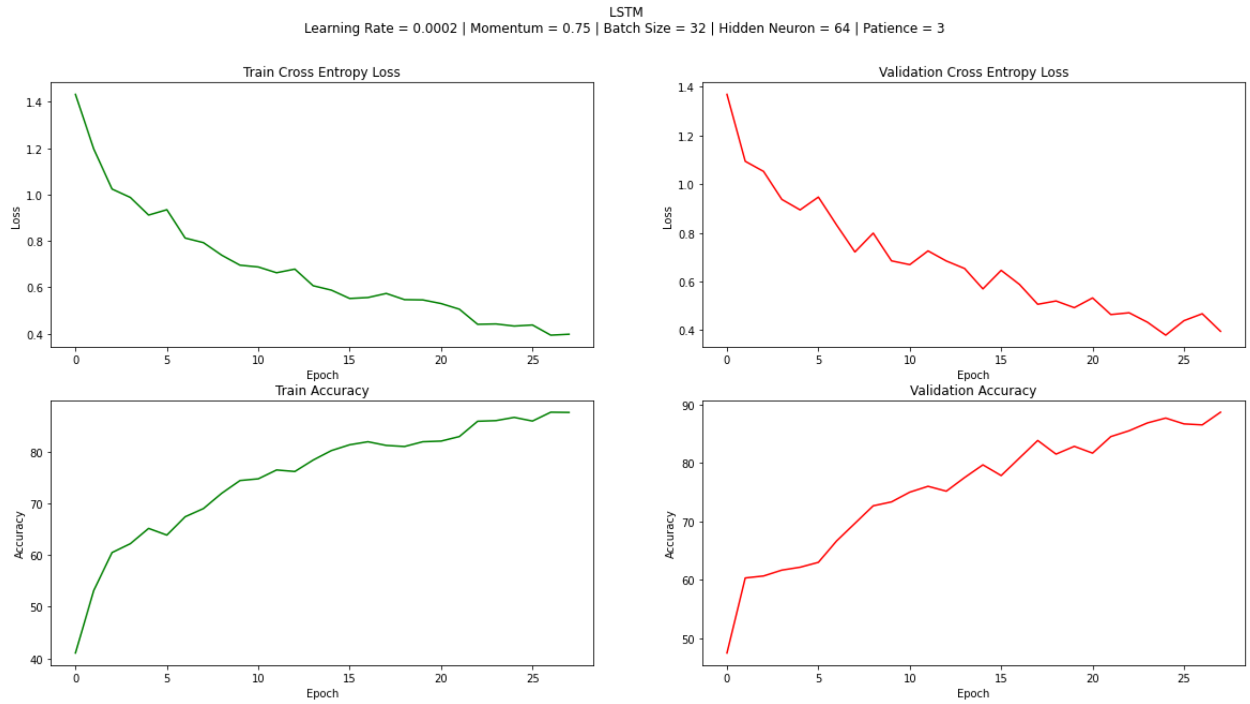
In the optimization of LSTM training, momentum with SGD is issued like section B. Start of each epoch training set is shuffled and separated 10 % for validation set. The parameters are adjusted to obtain high accuracy in validation and low cost at training and validation. Table 13 shows the parameters used in LSTM training process.

Learning Rate	0.0002
Momentum Rate	0.75
Epoch	100
Patience	3
Batch Size	32
Number of Hidden Neurons	64

Table 13: Training Parameters LSTM

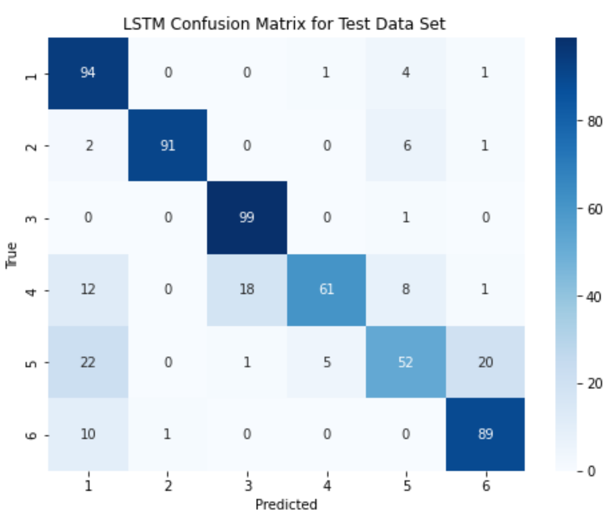
Train Cost	0.398
Validation Cost	0.396
Train Accuracy	90.76 %
Validation Accuracy	88.67 %
Test Accuracy	81.0 %

Table 14: Obtained Parameters LSTM

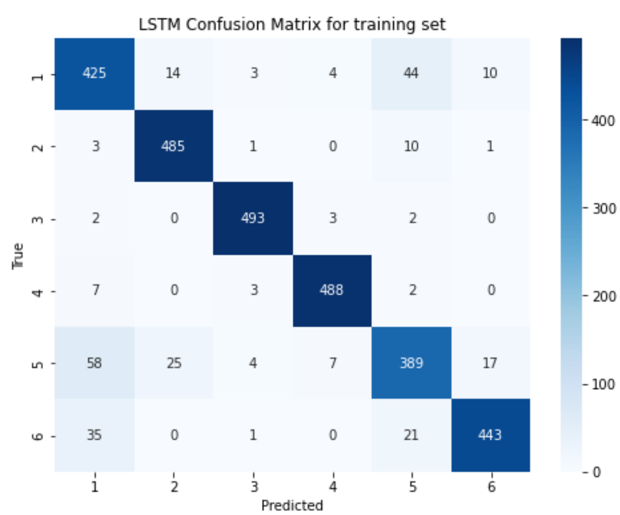


Graph 8: Graphs for Training Parameters for LSTM

In general trend of cross entropy loss of train and validation is decreasing. However, there is an oscillation in the validation accuracy and loss graph. Accuracy graphs of both cases are increasing which is same as expected.



Graph 9: Confusion Matrix Test for LSTM



Graph 10: Confusion Matrix Train for LSTM

Confusion Matrix of LSTM for training and test set shows model can predicted the output correctly generally. The test accuracy of LSTM is 81.0 %, so this model can consider as successful model since the test accuracy is near to validation and test accuracy.

The result obtained in LSTM is better than result obtained in RNN. There is nearly 24% percent increases in the test and train accuracy when LSTM is used. RNN model faces with vanishing gradient problem in longer time period. In longer period of time, RNN hidden layer activation function(tanh(x)) vanishes, so model learn long-term dependencies hardly. Gating structure helps to model do not saturate on higher time periods in LSTM. As a result, LSTM is more powerful in higher time period rather than RNN.

Section C

Moreover, the data is trained on alternative to LSTM and RNN, which is called gated recurrent units (GRU). GRU has two main gates update and reset gate. Update gate controls how much of the previous memory to keep and how much of the new information to add. Other gate is reset gate, which controls how much the past information to forget. The new hidden state is calculated as a combination of the previous state and the candidate hidden state. Graphical representation of GRU is

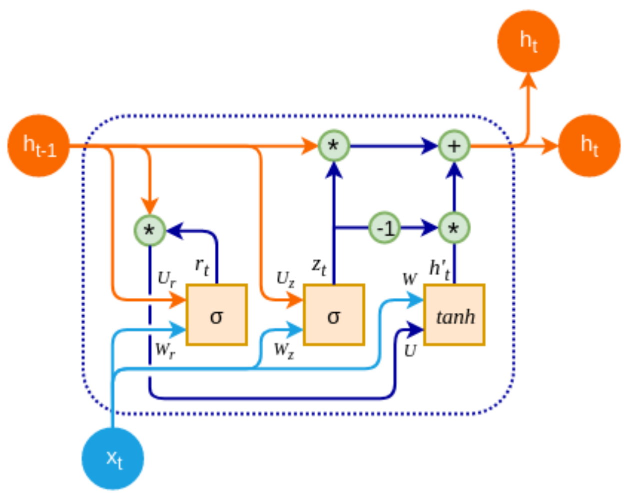


Fig. 11: GRU Structure [4]

In the forward propagation in GRU, there are several gates such as update and reset gate. The forward pass of these gates are shown below.

$$z_t = \sigma(W_z \cdot x_t + U_z \cdot h_{t-1} + b_z)$$

$$r_t = \sigma(W_r \cdot x_t + U_r \cdot h_{t-1} + b_r)$$

$$\tilde{h}t = \tanh(W_h \cdot x_t + U_h \cdot (r_t \odot ht - 1) + b_h)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h_t}$$

$$y_{\text{pred}} = \text{softmax}\left(h_T W_y + b_y\right)$$

The gradients of gates are calculated for BPTT of GRU.

$$\frac{\partial \mathcal{L}}{\partial z_{\text{output}}} = y_{\text{pred}} - y_{\text{true}}$$

$$\frac{\partial \mathcal{L}}{\partial W_y} = h_T^T \cdot \frac{\partial \mathcal{L}}{\partial z_{\text{output}}}$$

$$\frac{\partial \mathcal{L}}{\partial b_y} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial z_{\text{output}_i}}$$

$$\frac{\partial \mathcal{L}}{\partial h_T} = \frac{\partial \mathcal{L}}{\partial z_{\text{output}}} \cdot W_y^T$$

$$\delta \widetilde{h_t} = \frac{\partial \mathcal{L}}{\partial h_t} \cdot z_t$$

$$\frac{\partial \mathcal{L}}{\partial \tilde{h}t} = \delta \widetilde{h_t} \cdot \left(1 - \widetilde{h_t^2}\right)$$

$$\delta_{z_t} = \frac{\partial \mathcal{L}}{\partial h_t} \cdot \left(\tilde{h}t - ht - 1\right)$$

$$\frac{\partial \mathcal{L}}{\partial z_t} = \delta_{z_t} \cdot z_t \cdot (1 - z_t)$$

$$\delta_{r_t} = \frac{\partial \mathcal{L}}{\partial \tilde{h}t} \cdot (ht - 1U_h)$$

$$\frac{\partial \mathcal{L}}{\partial r_t} = \delta_{r_t} \cdot r_t \cdot (1 - r_t)$$

Gradient for Update Gate Parameters:

$$\frac{\partial \mathcal{L}}{\partial W_z} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial z_t} \cdot x_t^T$$

$$\frac{\partial \mathcal{L}}{\partial U_z} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial z_t} \cdot h_{t-1}^T$$

$$\frac{\partial \mathcal{L}}{\partial b_z} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial z_t}$$

Gradient for Reset Gate Parameters:

$$\frac{\partial \mathcal{L}}{\partial W_r} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial r_t} \cdot x_t^T$$

$$\frac{\partial \mathcal{L}}{\partial U_r} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial r_t} \cdot h_{t-1}^{T}$$

$$\frac{\partial \mathcal{L}}{\partial b_r} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial r_t}$$

Gradient for Candidate Hidden State:

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \tilde{h}t} \cdot x_t^{T}$$

$$\frac{\partial \mathcal{L}}{\partial U_h} = \sum t = 1^{T} \frac{\partial \mathcal{L}}{\partial \tilde{h}t} \cdot (r_t \odot ht - 1)^{T}$$

$$\frac{\partial \mathcal{L}}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \tilde{h}_t}$$

Additionally, the sigmoid is clipped in GRU owing to saturation.

```
def sigmoid(X):
    # Clip X to the range [-709, 709] to prevent overflow in exp
    X_clipped = np.clip(X, -709, 709)
    return 1 / (1 + np.exp(-X_clipped))
```

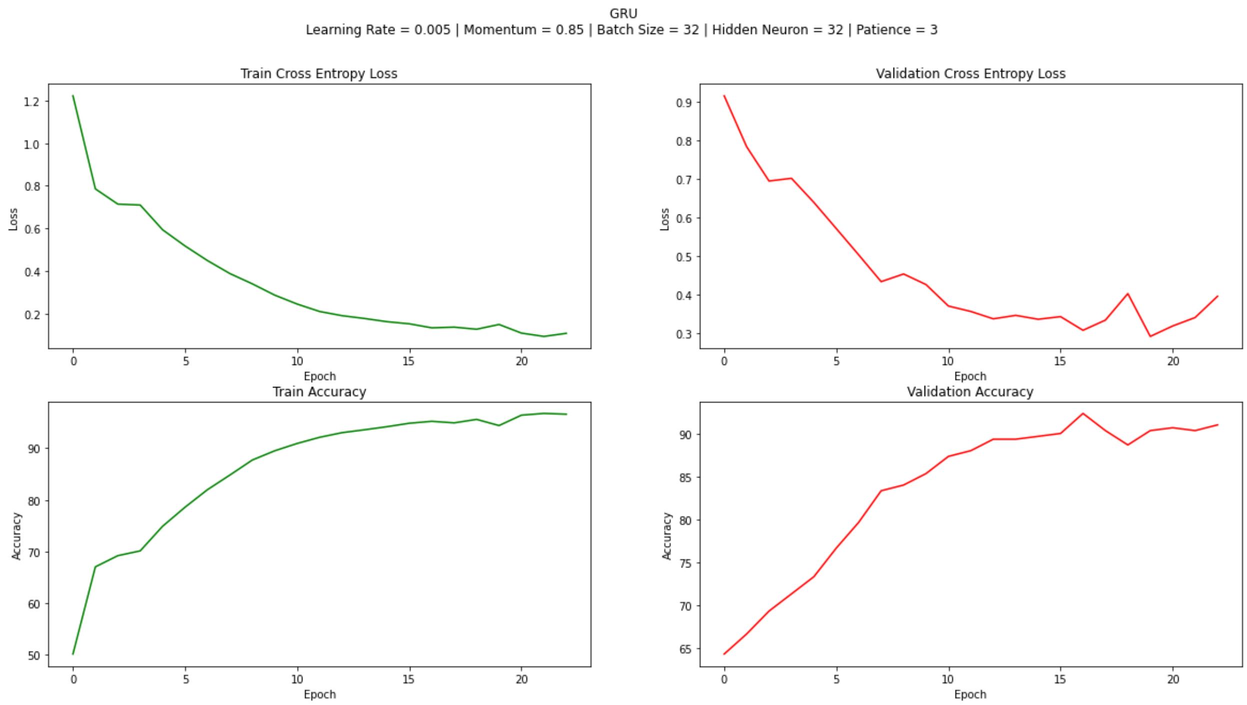
In the optimization of GRU training, momentum with SGD is issued like section B. The parameters are adjusted to obtain high accuracy in validation and low cost at training and validation. In the start of each epoch, train data set is shuffled and separated 10 % for validation. Table 15 shows the parameters used in GRU training process.

Learning Rate	0.005
Momentum Rate	0.85
Epoch	50
Patience	3
Batch Size	32
Number of Hidden Neuron	32

Table 15: Training Parameters GRU

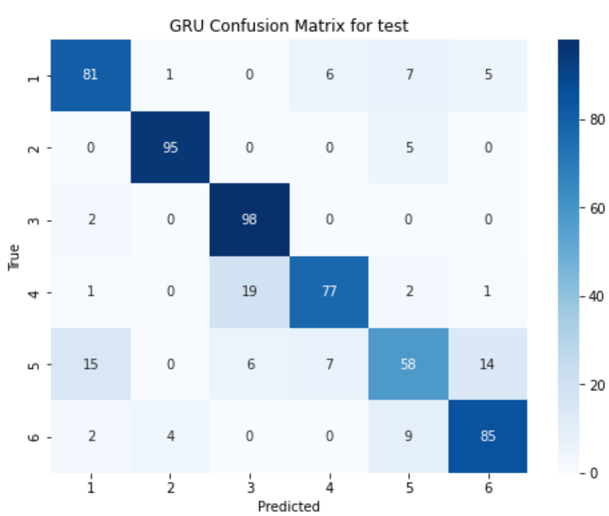
Train Cost	0.107
Validation Cost	0.395
Train Accuracy	96.54 %
Validation Accuracy	91.00 %
Test Accuracy	82.33 %

Table 16: Obtained Parameters GRU

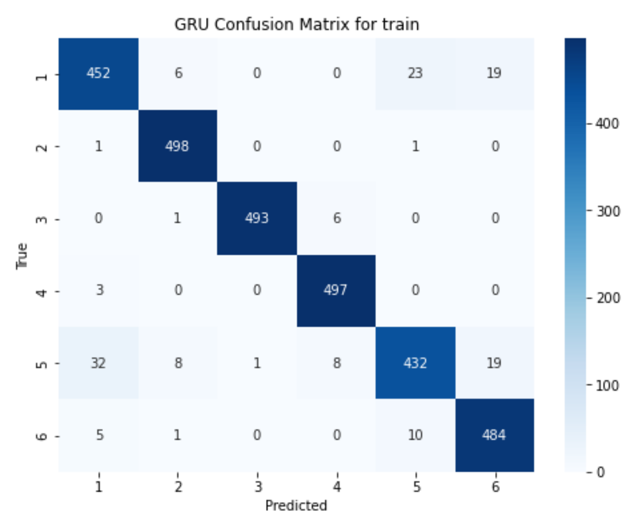


Graph 11: Graphs for Training Parameters for GRU

With training loss and accuracy exhibiting consistent improvement throughout epochs, the RNN's training process exhibits steady learning progress. The validation loss curve, on the other hand, shows minor oscillations albeit usually dropping, suggesting either overfitting or data noise. Validation accuracy rises gradually but reaches a plateau as the model gets closer to its generalization potential. By effectively halting training at the 24th epoch, overfitting is avoided and the model's capacity to generalize to new data is preserved.



Graph 12: Confusion Matrix Test for GRU



Graph 13: Confusion Matrix Train for GRU

The results obtained GRU can be considered as a success. GRU has fewer parameters compared to LSTM, makes it computationally efficient and faster in training. GRU and LSTM has 1.33 % difference in test accuracy. It is also better than RNN, since GRU does not have vanishing gradient problem. Consequently, GRU is capable of capture long term dependencies. Overall progress done in part 3, GRU has the highest test accuracy among the 3 sequential neural network structure because it is easier to implement and computationally faster.

	Test Accuracy	Training Time (seconds)	Final Epoch
RNN	57.33 %	27 seconds	9
LSTM	81.00 %	126 seconds	28
GRU	82.33 %	47 seconds	23

Table 17: Test Accuracy for Part 3

References

[1] "Recurrent Neural Networks," *IBM*. [Online]. Available: https://www.ibm.com/think/topics/recurrent-neural-networks. [Accessed: Dec. 5, 2024].

[2] "Many-to-one RNN structure used in the method of DNN with RNN on sequence," *ResearchGate*. [Online]. Available: https://www.researchgate.net/figure/Many-to-one-RNN-structure-used-in-the-method-of-DNN-with-RNNon-sequence_fig1_307889236. [Accessed: Dec. 5, 2024].

[3] "What is an LSTM Neural Network," DIDA. [Online]. Available: https://dida.do/what-is-an-lstm-neural-network/. [Accessed: Dec. 5, 2024].

[4] A. Nama, "Understanding Gated Recurrent Unit (GRU) in Deep Learning," Medium, [Online]. Available: https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2. [Accessed: Dec. 5, 2024].

Appendix

```
import numpy as np
import sys
import pandas as pd
import h5py
import math
import os
import matplotlib.pyplot as plt
import time
import random
import seaborn as sns

question = sys.argv[1]

def MustafaCankan_balci_22101761_hw1(question):
    if question == '1':
        # Question 1 code goes here
        print("Question 1 is selected!")
        # Open the file in read-only mode
        with h5py.File('data1.h5', 'r') as hdf:
            # List all groups and datasets in the file
            print("Keys:", list(hdf.keys()))

            # Access a specific group/dataset
            if 'your_dataset_key' in hdf:
                dataset = hdf['your_dataset_key']
                print("Dataset shape:", dataset.shape)
                print("Dataset dtype:", dataset.dtype)
            data = np.array(hdf['data'])

        # Preprocess the data by first converting the images to graysale usinga luminosity model: Y = 0.2126 ∗R+ 0.7152 ∗G+
0.0722 ∗B.

        Y = 0.2126*data[:,0,:,:] + 0.7152*data[:,1,:,:] + 0.0722*data[:,2,:,:]
        print(Y.shape)

        Y = np.reshape(Y, (Y.shape[0],256))  # flatten
        centered_Y = Y - Y.mean(axis=1, keepdims=True)

        standard_devition = centered_Y.std()
        clip_min = -3 * standard_devition
        clip_max = 3 * standard_devition
        images_clipped = np.clip(centered_Y, clip_min, clip_max)


        def normalize(X):
            """
            Normalizes given input
            @param X: input
            @return: normalized X
            """

            return (X - X.min())/(X.max() - X.min() + 1e-8)


        images_clipped = normalize(images_clipped)

        images_rescaled = 0.1 + images_clipped * 0.8
        print(images_rescaled.shape)
```

```
num_images,_ = Y.shape

images_grey = np.reshape(images_rescaled,(num_images,16,16))
num_patches = 200
patches_grey = []
patches_rgb = []

for _ in range(num_patches):
        img_idx = random.randint(0, num_images - 1)  # Random image index
        patch = images_grey[img_idx,:,:]
        rgb_patch = data[img_idx,:,:,:]
        patches_grey.append(patch)
        patches_rgb.append(rgb_patch)


patches_grey = np.array(patches_grey)
patches_rgb = np.array(patches_rgb)
patches_grey.shape

fig, axes = plt.subplots(10, 40, figsize=(20, 5))  # Display 10 patches per row for space
counter = 0
for j in range(5):

    for i in range(40):  # Display 10 patches at a time
        # Original RGB patch
        rgb_patch = np.transpose(patches_rgb[counter,:,:,:], (1, 2, 0))  # Convert (3, height, width) to (height, width, 3)
        rgb_patch = normalize(rgb_patch)
        axes[j, i].imshow(rgb_patch)
        axes[j, i].axis('off')

        # Normalized grayscale patch
        axes[5+j, i].imshow(patches_grey[counter,:,:], cmap='gray')
        axes[5+j, i].axis('off')
        counter += 1

plt.show()

# Part B

def initiliaze_weights(L_hid,L_pre,L_post,N):
    np.random.seed(3)

    w0 =  np.sqrt(6/(L_pre + L_post))

    W1 = np.random.uniform(-w0, w0, (N, L_hid))
    b1 = np.random.uniform(-w0, w0, (1,L_hid))
    W2 = W1.T
    b2 = np.random.uniform(-w0, w0, (1,N))


    return [W1,b1,W2,b2]

def init_momentum(L_hid,L_pre,L_post,N):

    W1 = np.zeros((N, L_hid))
    b1 = np.zeros((1,L_hid))
    W2 = W1.T
    b2 = np.zeros((1,N))

    momentum = {
        "W1": W1,
        "b1": b1,
```

```
        "W2": W2,
        "b2":  b2
    }

    return momentum

def sigmoid(W):
    return 1/(1+np.exp(-(W)))

def forward_prob(We,data):
    W1,b1, W2, b2 = We

    h1 = data @ W1 + b1
    h = sigmoid(h1)
    o1 = np.dot(h,W2) + b2
    o = sigmoid(o1)

    return {"Hidden Output":h, "Output of Autoencoder":o}


def aeCost(We,data,params):
    """
    Computes the cost and gradients for an autoencoder with sparsity constraints.

    Parameters:
    ----------
    We : list
        A list of weight and bias parameters: [W1, b1, W2, b2]
        - W1: Weight matrix for encoder (input_dim x hidden_dim)
        - b1: Bias vector for encoder (1 x hidden_dim)
        - W2: Weight matrix for decoder (hidden_dim x input_dim)
        - b2: Bias vector for decoder (1 x input_dim)
    data : ndarray
        Input data matrix (N x input_dim), where N is the number of samples.
    params : dict
        Dictionary containing the following keys:
        - "lambda" : float, regularization parameter for weight decay.
        - "rho" : float, desired average activation of the hidden units.
        - "beta" : float, weight of the sparsity penalty term.

    Returns:
    -------
    J : float
        The cost value for the autoencoder, including reconstruction error,
        regularization term, and sparsity penalty.
    dWe : list
        Gradients of the parameters [dW1, db1, dW2, db2], corresponding to the
        weight and bias matrices in `We`.

    Notes:
    -----
    - The sparsity penalty is based on the Kullback-Leibler divergence between
    the average activation of hidden units and the desired sparsity level `rho`.
    - Regularization is applied to the weights (W1 and W2) but not the biases.

    """

    lambda_reg = params["lambda"]
    rho = params["rho"]
    beta = params["beta"]
    N = data.shape[0]
```

```
    W1,b1,W2,b2= We
    outputs = forward_prob(We,data)
    y = outputs["Output of Autoencoder"]
    h = outputs["Hidden Output"]
    d = data

    # Calculate the cost
    rho_b = h.mean(axis=0, keepdims=True)  # 1 x Lhid


    b = np.sum(W1 ** 2)
    c = np.sum(W2 ** 2)


    KL = rho * np.log(rho/rho_b) + (1 - rho) * np.log((1 - rho)/(1 - rho_b))
    KL = beta * KL.sum()

    J = 0.5/N * (np.linalg.norm(d - y, axis=1) ** 2).sum() + 0.5 * lambda_reg * (b +  c) + KL


    # Encoder Backprob
    dy = (y-d)/N
    dy1 = y*(1-y)*dy
    dW2 = np.dot(h.T,dy1) + lambda_reg*W2
    db2 = dy1.sum(axis=0,keepdims=True)

    # Without KL Divergence Decoder
    #print(W1.shape)
    dpj = beta*(-rho/rho_b + (1-rho)/(1-rho_b))/N
    dh = np.dot(dy1,W2.T) + dpj
    dh1 = h*(1-h)*dh
    dW1 = np.dot(d.T,dh1) + lambda_reg*W1
    db1 = dh1.sum(axis=0,keepdims=True)


    dW2 = (dW1.T + dW2)
    dW1 = dW2.T

    dWe = [dW1, db1, dW2, db2]

    return J,dWe

def solver(grads,We,learning_rate,momentum,alpha):
    W1,b1,W2,b2 = We
    dW1,db1,dW2,db2 =  grads

    momentum["W1"] = alpha *momentum["W1"] - learning_rate*dW1
    W1 = W1 + momentum["W1"]
    momentum["b1"] = alpha *momentum["b1"] - learning_rate*db1
    b1 = b1 + momentum["b1"]
    momentum["W2"] = alpha *momentum["W2"] - learning_rate*dW2
    W2 = W2 + momentum["W2"]
    momentum["b2"] = alpha *momentum["b2"] - learning_rate*db2
    b2 = b2 + momentum["b2"]

    return [W1,b1,W2,b2],momentum

def train_nn_q1(epoch_number,X,batch_size,We,momentum,learning_rate,alpha,params):
    J_list = []
    for i in range(0, epoch_number):
        np.random.seed(3)
```

```
        start = 0
        end = batch_size
        J_temp = 0
        start_time = time.time()
        # Shuffle data set
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)

        X = X[indices]

        num_it =  X.shape[0] // batch_size


        for j in range(num_it):
            batch_data = X[start:end]
            np.random.seed(3)
            indices = np.arange(batch_data.shape[0])
            np.random.shuffle(indices)

            batch_data = batch_data[indices]

            J, J_grads = aeCost(We,batch_data,params)
            We,momentum = solver(J_grads,We,learning_rate,momentum,alpha)
            J_temp += J
            start = end
            end += batch_size
        J_list.append((J_temp/num_it))
        end_time = time.time()
        elasped_time = end_time-start_time
        print(f'Epoch {i} cost: {J_temp/num_it} time : {elasped_time:.2f}')


    return We,J_list

num_iterations = 200
L_hid  = 64
L_pre = 256
L_post = L_hid
N = 256
learning_rate = 0.01
batch_size = 32
alpha = 0.85


Y_reshape = images_rescaled

We = initiliaze_weights(L_hid,L_pre,L_post,N)
momentum = init_momentum(L_hid,L_pre,L_post,N)

lambda_part_b = 5e-4

params = {"Lin":64,"Lhid":L_hid,"beta":1,"rho":0.04,"lambda":lambda_part_b}


J_list = []


We,J_list = train_nn_q1(num_iterations,images_rescaled,batch_size,We,momentum,learning_rate,alpha,params)

plt.title("Cost function when beta = 1 and rho = 0.04")
plt.plot(J_list)
plt.xlabel("epoch")
```

```
plt.ylabel("cost")
plt.show()

c = forward_prob(We,images_rescaled)

images_grey = np.reshape(images_rescaled,(num_images,16,16))

b= c["Output of Autoencoder"][100].reshape(16,16)


plt.imshow(b, cmap='gray')
plt.title("Output Image of Autoencoder")
plt.axis('off')
plt.show()

plt.imshow(images_grey[100], cmap='gray')
plt.title("Input Image")
plt.axis('off')
plt.show()

# Section C
def plot_hidden_weight(We,parameters):

    counter = 0

    W1,b1,W2,b2= We
    W1_reshaped = W1.T
    W1_reshaped = W1_reshaped.reshape(W1_reshaped.shape[0],16,16)
    number_hidden = W1_reshaped.shape[0]

    if number_hidden == 64:
        l = 8
        k = 8
        fig, axes = plt.subplots(8, 8)  # Display 10 patches per row for space
    elif number_hidden == 100:
        l = 10
        k = 10
        fig, axes = plt.subplots(10, 10)  # Display 10 patches per row for space
    elif number_hidden == 10:
        l = 2
        k = 5
        fig, axes = plt.subplots(2, 5)  # Display 10 patches per row for space


    for j in range(l):
        for i in range(k):
        #Normalized grayscale patch
            axes[j, i].imshow(W1_reshaped[counter,:,:], cmap='gray')
            axes[j, i].axis('off')
            counter += 1
    b = parameters["beta"]
    rho = parameters["rho"]
    lambda_val = parameters["lambda"]
    fig.suptitle(
        f"Beta = {b} when rho = {rho}  number hidden: {l*k} lambda: {lambda_val}  ",
        fontsize=12
        )

    plt.show()

plot_hidden_weight(We,params)
```

```
    # Section D

    lambda_val_list = [0,1e-3,5e-4]
    L_hid_list = [10,100,64]

    weigths = []
    J_list_it = []

    for i in lambda_val_list:
        for j in L_hid_list:
            # Creating Pre values
            num_iterations = 200
            batch_size = 32
            learning_rate = 0.01
            J_list = []

            alpha = 0.85
            Y_reshape = images_rescaled


            We = initiliaze_weights(j,L_pre,L_post,N)
            momentum = init_momentum(j,L_pre,L_post,N)

            params = {"Lhid":j,"beta":1,"rho":0.04,"lambda":i}

            # Train the model
            print(f"L_hid: {j} lambda: {i} Epochs")
            We,J_list =  train_nn_q1(num_iterations,images_rescaled,batch_size,We,momentum,learning_rate,alpha,params)
            weigths.append(We)
            J_list_it.append(J_list)

            print(f"L_hid: {j} lambda: {i} Graph")
            # display hidden layers features
            plot_hidden_weight(We,params)


elif question == '2':
    # Question 2 code goes here
    print("Question 2 is selected")

    # Uploading Parameters
    # Open the file in read-only mode
    with h5py.File('data2.h5', 'r') as hdf:
        # List all groups and datasets in the file
        print("Keys:", list(hdf.keys()))

        # Access a specific group/dataset
        if 'your_dataset_key' in hdf:
            dataset = hdf['your_dataset_key']
            print("Dataset shape:", dataset.shape)
            print("Dataset dtype:", dataset.dtype)
        test_d = np.array(hdf["testd"])
        test_x = np.array(hdf["testx"])
        train_d = np.array(hdf["traind"])
        train_x = np.array(hdf["trainx"])
        val_d = np.array(hdf["vald"])
        val_x = np.array(hdf["valx"])
        words = np.array(hdf["words"])

    def init_weigths(D,P):
        W_ih = np.random.normal(0, 0.01, (D*3,P))
        b_ih = np.random.normal(0, 0.01, (1,P))
```

```
    W_ho = np.random.normal(0, 0.01, (P,250))
    b_ho = np.random.normal(0, 0.01, (1,250))
    W_embedded_weights = np.random.normal(0, 0.01, (250,D))

    parameters = {
        "W_ih": W_ih,
        "b_ih": b_ih,
        "W_ho": W_ho,
        "b_ho": b_ho,
        "W_embedded_weights": W_embedded_weights
    }
    return parameters

def init_momentum(D,P):
    W_ih = np.zeros((D*3,P))
    b_ih = np.zeros((1,P))
    W_ho = np.zeros((P,250))
    b_ho = np.zeros((1,250))
    W_embedded_weights = np.zeros((250,D))

    momentum = {
        "dW_ih": W_ih,
        "db_ih": b_ih,
        "dW_ho": W_ho,
        "db_ho": b_ho,
        "dW_embedded_weights": W_embedded_weights
    }
    return momentum

def softmax(x):
     """Compute softmax values for each set of scores in x."""
     exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))  # For numerical stability
     return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def sigmoid(Z):
    return 1/(1+ np.exp(-Z))

def forward_pass(parameters,list_embeeded):
    W_ih = parameters["W_ih"]
    b_ih = parameters["b_ih"]
    W_ho = parameters["W_ho"]
    b_ho = parameters["b_ho"]
    W_embedded_weights = parameters["W_embedded_weights"]

    embedding1 =  W_embedded_weights[list_embeeded[:,0]-1,:]
    embedding2 = W_embedded_weights[list_embeeded[:,1]-1,:]
    embedding3 = W_embedded_weights[list_embeeded[:,2]-1,:]

    embeddings = np.concatenate([embedding1, embedding2, embedding3], axis=1)  # Shape: (batch_size, embedding_dim *
3)

    hidden_output_activation= np.dot(embeddings ,W_ih) + b_ih
    hidden_output = sigmoid(hidden_output_activation)


    softmax_output_activation= np.dot(hidden_output,W_ho) + b_ho
    output = softmax(softmax_output_activation)

    return embeddings,hidden_output,output

def cross_entropy_loss(desired,output):
    desired_one_hot = []
```

```
    # Ensure desired is one-hot encoded
    m = desired.shape[0]
    for i in range(m):
        encoded_arr = np.zeros((250,), dtype=int)
        encoded_arr[desired[i]-1] = 1
        desired_one_hot.append(encoded_arr)
    desired_one_hot = np.array(desired_one_hot)

    # Cross-entropy loss
    cost = -np.sum(desired_one_hot * np.log(output))/m
    return cost

def backpropagation(parameters,y,d,h,embeddings,x):
    W_ih = parameters["W_ih"]
    b_ih = parameters["b_ih"]
    W_ho = parameters["W_ho"]
    b_ho = parameters["b_ho"]
    W_embedded_weights = parameters["W_embedded_weights"]

    desired_one_hot = []

    m = y.shape[0]

    # One-hot encoding (vectorized)
    desired_one_hot = np.zeros_like(y)
    desired_one_hot[np.arange(m), d - 1] = 1  # Assuming d starts at 1


    # Finding derivatives
    dZ = y - desired_one_hot
    dAo =  dZ

    dW_ho =  np.dot(h.T, dAo)/m
    db_ho = np.sum(dAo,axis = 0,keepdims=True)/m

    d_h = np.dot(dAo, W_ho.T )
    dA2 =  h*(1-h)* d_h
    dW_ih =  np.dot(embeddings.T ,dA2)/m
    db_ih = np.sum(dA2,axis = 0,keepdims=True) /m


    # Gradients for embedding layer
    d_embeddings = np.dot(dA2, W_ih.T)/m
    dW_embedded_weights = np.zeros_like(W_embedded_weights)

    # Accumulate embedding gradients for each trigram word
    for i in range(3):  # Each trigram has 3 words
        np.add.at(dW_embedded_weights, x[:, i]-1, d_embeddings[:, i::3])


    grads = {
        "dW_ih": dW_ih,
        "db_ih": db_ih,
        "dW_ho": dW_ho,
        "db_ho": db_ho,
        "dW_embedded_weights": dW_embedded_weights
    }


    return grads

def update_parameters(parameters, momentum, grads, alpha, learning_rate):
```

```
    # Update momentum terms and parameters for W_ho
    momentum["dW_ho"] = alpha * momentum["dW_ho"] - learning_rate * grads["dW_ho"]
    parameters["W_ho"] += momentum["dW_ho"]

    # Update momentum terms and parameters for b_ho
    momentum["db_ho"] = alpha * momentum["db_ho"] - learning_rate * grads["db_ho"]
    parameters["b_ho"] += momentum["db_ho"]

    # Update momentum terms and parameters for W_ih
    momentum["dW_ih"] = alpha * momentum["dW_ih"] - learning_rate * grads["dW_ih"]
    parameters["W_ih"] += momentum["dW_ih"]

    # Update momentum terms and parameters for b_ih
    momentum["db_ih"] = alpha * momentum["db_ih"] - learning_rate * grads["db_ih"]
    parameters["b_ih"] += momentum["db_ih"]

    # Update momentum terms and parameters for W_embedded_weights
    momentum["dW_embedded_weights"] = alpha * momentum["dW_embedded_weights"] - learning_rate *
grads["dW_embedded_weights"]
    parameters["W_embedded_weights"] += momentum["dW_embedded_weights"]

    return parameters, momentum

def compute_accuracy(y_pred, y_true):
    """
    Computes the accuracy given predictions and true labels.

    Args:
        y_pred (numpy.ndarray): Predicted probabilities, shape (num_samples, num_classes).
        y_true (numpy.ndarray): True labels, shape (num_samples,).

    Returns:
        float: Accuracy as a percentage.
    """
    predictions = np.argmax(y_pred, axis=1) + 1  # +1 to match label indexing starting at 1
    accuracy = np.mean(predictions == y_true) * 100
    return accuracy

def run_question_2(D,P):

    parameters = init_weigths(D,P)
    momentum = init_momentum(D,P)

    learning_rate = 0.15
    alpha = 0.85
    batch_size = 200
    epochs = 50

    patience = 5

    best_val_cost = float('inf')
    epoch_interpt = 0

    total_time = 0

    val_acc_list = []
    val_cost_list = []
    train_cost_list = []
    train_acc_list = []

    for epoch_n in range(epochs):
```

```
np.random.seed(42)

iteration_each_epoch = train_x.shape[0] // batch_size

start_time = time.time()
start = 0
end = batch_size
cost = 0

# Shuffle data set
indices = np.arange(train_x.shape[0])
np.random.shuffle(indices)

train_x_shuf = train_x[indices]
train_d_shuf = train_d[indices]
train_acc = 0


for j in range(iteration_each_epoch):

    data_temp = train_x_shuf[start:end]
    result_temp = train_d_shuf[start:end]


    # Forward Pass
    embeddings,hidden,output = forward_pass(parameters,data_temp)

    # Find Cost and Backprob
    cost += cross_entropy_loss(result_temp,output)

    temp_acc = compute_accuracy(output,result_temp)

    train_acc += temp_acc

    grads = backpropagation(parameters,output,result_temp,hidden,embeddings,data_temp)

    # Update the Result
    parameters,momentum = update_parameters(parameters,momentum,grads,alpha,learning_rate)

    start = end
    end += batch_size

end_time = time.time()
elapsed_time = end_time - start_time
total_time += elapsed_time

total_cost_train = cost/iteration_each_epoch
train_acc /= iteration_each_epoch


_,_,val_out= forward_pass(parameters,val_x)
val_cost = cross_entropy_loss(val_d,val_out)

val_cost = np.round(val_cost, 3)
total_cost_train = np.round(total_cost_train, 3)

if val_cost < best_val_cost:
    best_val_cost = val_cost
    epoch_interpt = 0
else:
    epoch_interpt += 1
```

```
        val_acc = compute_accuracy(val_out,val_d)

        print(f'Epoch {epoch_n}: Train Los Cost: {total_cost_train} and Validation Cost: {val_cost}, Train Acc: {train_acc}
and   Validation Acc: {val_acc} time taken to train: {int(elapsed_time)} s'  )
        train_cost_list.append(total_cost_train)
        val_acc_list.append(val_acc)
        val_cost_list.append(val_cost)
        train_acc_list.append(train_acc)

        cost = 0

        if epoch_interpt >= patience:
            print("Early Stopped!")
            break
    return parameters, train_cost_list, val_cost_list, val_acc_list, train_acc_list

sizes = [[32,256], [16,128],[8, 64]]

store_values = []
train_cost_list = []
val_cost_list = []
val_acc_list = []
train_acc_list = []

for value in sizes:
    D = value[0]
    P = value[1]
    # Run Neural Netwrok
    print(f"D : {D} and P: {P}")
    parameters,train_cost_list_i, val_cost_list_i, val_acc_list_i,train_acc_list_i  = run_question_2(D,P)
    store_values.append(parameters)
    train_cost_list.append(train_cost_list_i)
    val_acc_list.append(val_acc_list_i)
    val_cost_list.append(val_cost_list_i)
    train_acc_list.append(train_acc_list_i)


# Plot the images
# Create subplots

def plotpart2sectiona(counter):
    sizes = [[32,256], [16,128],[8, 64]]
    fig, axs = plt.subplots(2, 2, figsize=(20, 10))

    # Train loss plot
    axs[0, 0].plot(train_cost_list[counter], color='green')
    axs[0, 0].set_title('Train Cross Entropy Loss')
    axs[0, 0].set_xlabel('Epoch')
    axs[0, 0].set_ylabel('Loss')

    # Validation loss plot
    axs[0, 1].plot(val_cost_list[counter], color='red')
    axs[0, 1].set_title('Validation Cross Entropy Loss')
    axs[0, 1].set_xlabel('Epoch')
    axs[0, 1].set_ylabel('Loss')

    # Train accuracy plot
    axs[1, 0].plot(train_acc_list[counter], color='green')
    axs[1, 0].set_title('Train Accuracy')
    axs[1, 0].set_xlabel('Epoch')
    axs[1, 0].set_ylabel('Accuracy')
```

```
    # Validation accuracy plot
    axs[1, 1].plot(val_acc_list[counter], color='red')
    axs[1, 1].set_title('Validation Accuracy')
    axs[1, 1].set_xlabel('Epoch')
    axs[1, 1].set_ylabel('Accuracy')

    # Add an overall title
    fig.suptitle(
      f"Part 2\nLearning Rate = 0.15 | Momentum = 0.85 | Batch Size = 200 | Patience = 5 | D = {sizes[counter][0]} and  P = 
{sizes[counter][1]} ",
      fontsize=12
    )


    plt.show()

  for i in range(3):
    plotpart2sectiona(i)



  print("Part B")

  random_sample_list_x = []
  random_sample_list_d = []


  for j in range(5):
    random_sample_2 = random.randint(0, test_x.shape[0])
    random_sample_list_x.append(test_x[random_sample_2])
    random_sample_list_d.append(test_d[random_sample_2])

  e,hidden_val,output_test  = forward_pass(store_values[0],np.array(random_sample_list_x))

  counter = 1
  for i in random_sample_list_x:
    sentence  = ""
    score = []
    for j in range(len(i)):
      b = len(str(words[i[j]-1]))
      sentence += str(words[i[j]-1])[2:b-1]
      sentence += " "
    print(f"{counter} Sample in List")
    print(sentence)
    # Selecting 10 highest values

    ten_high_value  = []
    print("Ten highest value at the output of neural network:")
    for i in range(10):
      if i == 0:
        ten_high_value.append(words[np.where(output_test[counter -1] == np.max(output_test[counter-1]))[0][0]])
        score.append(np.max(output_test[counter-1]))
        temp_list = np.delete(output_test[counter-1],np.where(output_test[counter -1] == np.max(output_test[counter-
1]))[0][0])
      else:
        ten_high_value.append(words[np.where(temp_list == np.max(temp_list))[0][0]])
        score.append(np.max(temp_list))
        temp_list = np.delete(temp_list,np.where(temp_list == np.max(temp_list))[0][0])



    print(ten_high_value)
```

```
        print("Scores:")
        print(np.array(score))
        print("Desired word: ")
        print(words[random_sample_list_d[counter -1]-1])



        counter += 1


elif question == '3':
    # Question 3 code goes here
    print("Question 3 is selected")

    # Open the file in read-only mode
    with h5py.File('data3.h5', 'r') as hdf:
        # List all groups and datasets in the file
        print("Keys:", list(hdf.keys()))

        # Access a specific group/dataset
        if 'your_dataset_key' in hdf:
            dataset = hdf['your_dataset_key']
            print("Dataset shape:", dataset.shape)
            print("Dataset dtype:", dataset.dtype)
        trX = np.array(hdf["trX"])
        trY = np.array(hdf["trY"])
        tstX = np.array(hdf["tstX"])
        tstY = np.array(hdf["tstY"])

    # Part A

    def softmax(x):
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def sigmoid(X):
        return 1/ (1+ np.exp(-X))

    def RNN_init_weights():
        np.random.seed(38)

        # Input size, hidden size, and output size
        input_size = 3
        hidden_size = 128
        output_size = 6

        # Xavier Uniform Initialization for weights
        limit_W_xh = np.sqrt(6 / (input_size + hidden_size))
        limit_W_hh = np.sqrt(6 / (hidden_size + hidden_size))
        limit_W_ho = np.sqrt(6 / (hidden_size + output_size))

        W_xh = np.random.uniform(-limit_W_xh, limit_W_xh, (input_size, hidden_size))
        W_hh = np.random.uniform(-limit_W_hh, limit_W_hh, (hidden_size, hidden_size))
        W_ho = np.random.uniform(-limit_W_ho, limit_W_ho, (hidden_size, output_size))

        # Biases
        # Biases are often initialized to zero
        b_x = np.random.uniform(-limit_W_xh, limit_W_xh,(1, hidden_size))
        b_o = np.random.uniform(-limit_W_ho, limit_W_ho,(1, output_size))

        parameters = {
            "W_hh": W_hh,
```

```
        "b_x": b_x,
        "W_xh": W_xh,
        "W_ho": W_ho,
        "b_o": b_o
    }

    return parameters

def RNN_forward_pass(xt, h_prev, parameters):

    W_xh = parameters["W_xh"]
    W_hh = parameters["W_hh"]
    W_ho = parameters["W_ho"]
    b_x = parameters["b_x"]
    b_o = parameters["b_o"]


    h = np.tanh(np.dot(xt,W_xh) + np.dot(h_prev,W_hh) +b_x)
    y = softmax(np.dot(h,W_ho) + b_o)

    return y,h

def RNN_forward_t_times(parameters,x):

    # Store previous values

    n_samples, time, n_features = x.shape
    # Initalize values
    h_prev = np.zeros((n_samples,128))

    output_list = []
    hidden_list = []

    for t in range(time):
        y_out ,h_out = RNN_forward_pass(x[:,t,:],h_prev,parameters)
        h_prev = h_out

        # adding obtained output in the list
        output_list.append(y_out)
        hidden_list.append(h_prev)

    return np.array(output_list),np.array(hidden_list)

def init_momentum_weight():
    input_size = 3
    hidden_size = 128
    output_size = 6

    dW_xh = np.zeros((input_size, hidden_size))
    dW_hh = np.zeros((hidden_size, hidden_size))
    dW_ho = np.zeros((hidden_size, output_size))


    # Biases are often initialized to zero
    db_x = np.zeros((1, hidden_size))
    db_o = np.zeros((1, output_size))

    momentum = {
        "dW_hh": dW_hh,
        "db_x": db_x,
        "dW_xh": dW_xh,
        "dW_ho": dW_ho,
```

```
        "db_o": db_o
    }
    return momentum

def cost_rnn(y_true, y_pred):
    """
    y_true: (n_samples, output_size), one-hot encoded
    y_pred: (n_samples, output_size), predicted probabilities
    """
    m = y_true.shape[0]
    # Cross-entropy loss
    cost = -np.sum(y_true * np.log(y_pred + 1e-12)) / m
    return cost


def bptt(parameters, X, hidden, y_true, y_pred):
    """
    Perform Backpropagation Through Time.

    X: (n_samples, time_steps, input_size)
    hidden: (time_steps, n_samples, hidden_size)
    y_true: (n_samples, output_size) ground truth at final timestep
    y_pred: (n_samples, output_size) predictions at final timestep

    Assumption: The loss is computed at the final timestep only.
    """

    W_xh = parameters["W_xh"]
    W_hh = parameters["W_hh"]
    W_ho = parameters["W_ho"]

    time_steps = hidden.shape[0]
    n_samples = X.shape[0]

    dW_ho = np.zeros_like(W_ho)
    dW_hh = np.zeros_like(W_hh)
    dW_xh = np.zeros_like(W_xh)
    db_x = np.zeros_like(parameters["b_x"])
    db_o = np.zeros_like(parameters["b_o"])

    # Gradient of loss wrt output logits at final timestep
    # y_pred: (n_samples, output_size)
    # y_true: (n_samples, output_size)
    dZo = (y_pred - y_true)  # shape: (n_samples, output_size)

    # Gradients wrt W_ho and b_o
    # Use the final hidden state: hidden[-1,:,:] shape (n_samples, hidden_size)
    final_h = hidden[-1, :, :]
    dW_ho = np.dot(final_h.T, dZo)
    db_o = np.sum(dZo, axis=0, keepdims=True)

    # Backprop through time
    dh_next = np.zeros((n_samples, W_hh.shape[0]))

    for t in reversed(range(time_steps)):
        # For the final timestep, include gradient from output layer
        if t == time_steps - 1:
            dA = np.dot(dZo, W_ho.T) + dh_next
        else:
            # No direct output gradient at earlier timesteps if we're only
            # considering loss at the final timestep
            dA = dh_next
```

```
        # dtanh = dA * (1 - h^2)
        h_t = hidden[t, :, :]  # (n_samples, hidden_size)
        dtanh = dA * (1 - h_t**2)

        # Gradients wrt W_xh and W_hh, b_x
        # X[:, t, :] shape: (n_samples, input_size)
        dW_xh += np.dot(X[:, t, :].T, dtanh)
        db_x += np.sum(dtanh, axis=0, keepdims=True)

        if t > 0:
            h_prev = hidden[t-1, :, :]
            dW_hh += np.dot(h_prev.T, dtanh)
        else:
            # At the first timestep, there's no previous hidden state from hidden array
            # If the initial hidden state is always zero, the contribution is zero.
            pass

        # Compute dh_next for the next iteration
        dh_next = np.dot(dtanh, W_hh.T)

    grads = {
        "dW_ho": dW_ho,
        "dW_hh": dW_hh,
        "dW_xh": dW_xh,
        "db_x": db_x,
        "db_o": db_o
    }

    return grads

def update_parameters(parameters, momentum, grads, alpha, learning_rate):
    # alpha is momentum factor
    # learning_rate is step size
    for param_name in ["W_ho", "W_hh", "W_xh", "b_o", "b_x"]:
        dparam_name = "d" + param_name
        momentum[dparam_name] = alpha * momentum[dparam_name] - learning_rate * grads[dparam_name]
        parameters[param_name] += momentum[dparam_name]

    return parameters, momentum

def compute_accuracy(y_pred, y_true):
    """
    Computes the accuracy given predictions and true labels.
    Assumes y_true is one-hot encoded.
    """
    pred_labels = np.argmax(y_pred, axis=1)
    true_labels = np.argmax(y_true, axis=1)
    accuracy = np.mean(pred_labels == true_labels) * 100
    return accuracy

# Train the model

# Initalize Parameters
alpha = 0.95
learning_rate = 0.000015
mini_batch = 32
epoch_size = 40

patience = 2

best_val_cost = float('inf')
```

```
epoch_interpt = 0

total_time = 0

# initliaze weights
momentum = init_momentum_weight()
parameters = RNN_init_weights()

train_cost_list_rnn = []
val_cost_list_rnn = []
train_acc_list_rnn = []
val_acc_list_rnn = []


for epoch in range(epoch_size):

    start_time = time.time()
    np.random.seed(38)

    # Shuffle dataset
    indices = np.arange(trX.shape[0])
    np.random.shuffle(indices)
    trx_shuffle_x = trX[indices, :, :]
    trx_shuffle_y = trY[indices, :]

    # Split into train/val
    length_tr = trX.shape[0]
    train_set_length = int(length_tr * 0.9)

    train_X = trx_shuffle_x[:train_set_length,:,:]
    train_Y = trx_shuffle_y[:train_set_length,:]

    val_X = trx_shuffle_x[train_set_length:,:,:]
    val_Y = trx_shuffle_y[train_set_length:,:]

    start = 0
    end = mini_batch
    temp_cost = 0
    train_acc = 0

    num_batches = train_X.shape[0] // mini_batch


    for j in range(num_batches):
        batch_x = train_X[start:end]
        batch_y = train_Y[start:end]

        # Forward Pass (over all timesteps)
        output_list, hidden_list = RNN_forward_t_times(parameters, batch_x)

        # output_list: (time_steps, batch_size, output_size)
        # We assume loss at the final timestep only
        final_output = output_list[-1, :, :]  # shape: (batch_size, output_size)

        train_acc += compute_accuracy(final_output, batch_y)

        # Calculate cost
        cost = cost_rnn(batch_y, final_output)
        temp_cost += cost

        # Get gradients
        grads = bptt(parameters, batch_x, hidden_list, batch_y, final_output)
```

```
        # Update Parameters
        parameters, momentum = update_parameters(parameters, momentum, grads, alpha, learning_rate)

        start = end
        end += mini_batch

    # Validation
    val_out, _ = RNN_forward_t_times(parameters, val_X)
    val_final = val_out[-1, :, :]  # final timestep predictions
    val_cost = cost_rnn(val_Y, val_final)
    val_acc = compute_accuracy(val_final, val_Y)

    total_cost_train = temp_cost / num_batches
    train_acc_total = train_acc / num_batches

    end_time = time.time()
    elapsed_time = end_time - start_time

    print(f'Epoch {epoch +1 }: Train Cost: {total_cost_train:.3f}, Train Acc: {train_acc_total:.2f}% '
        f'Val Cost: {val_cost:.3f}, Val Acc: {val_acc:.2f}%, '
        f'Time: {int(elapsed_time)}s  patience: {epoch_interpt}')

    # Saving Parameters
    train_cost_list_rnn.append(total_cost_train)

    val_cost_list_rnn.append(val_cost)

    train_acc_list_rnn.append(train_acc_total)

    val_acc_list_rnn.append(val_acc)

    # Early stopping
    if val_cost < best_val_cost:
        best_val_cost = val_cost
        epoch_interpt = 0
    else:
        epoch_interpt += 1
        if epoch_interpt >= patience:
            print("Early Stopped!")
            break

def plot_confusion_matrix(y_true, y_pred, class_names,title):
    cm = np.zeros((len(class_names), len(class_names)), dtype=int)
    for t, p in zip(y_true, y_pred):
        cm[t, p] += 1
    plt.figure(figsize=(8,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
    plt.title(title)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()
    return cm

# Write Code for Plotting the Graph
# Train cost Train Acc Validation Cost Validation Accuracy


fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Train loss plot
axs[0, 0].plot(train_cost_list_rnn, color='green')
```

```
axs[0, 0].set_title('Train Cross Entropy Loss')
axs[0, 0].set_xlabel('Epoch')
axs[0, 0].set_ylabel('Loss')

# Validation loss plot
axs[0, 1].plot(val_cost_list_rnn, color='red')
axs[0, 1].set_title('Validation Cross Entropy Loss')
axs[0, 1].set_xlabel('Epoch')
axs[0, 1].set_ylabel('Loss')

# Train accuracy plot
axs[1, 0].plot(train_acc_list_rnn, color='green')
axs[1, 0].set_title('Train Accuracy')
axs[1, 0].set_xlabel('Epoch')
axs[1, 0].set_ylabel('Accuracy')

# Validation accuracy plot
axs[1, 1].plot(val_acc_list_rnn, color='red')
axs[1, 1].set_title('Validation Accuracy')
axs[1, 1].set_xlabel('Epoch')
axs[1, 1].set_ylabel('Accuracy')

# Add an overall title
fig.suptitle(
   "RNN\nLearning Rate = 0.000015 | Momentum = 0.95 | Batch Size = 32 | Hidden Neuron = 128 ",
   fontsize=12
)

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()


# Forward Pass (over all timesteps)
output_test, _ = RNN_forward_t_times(parameters, tstX)

# output_list: (time_steps, batch_size, output_size)
# We assume loss at the final timestep only
final_test_output = output_test[-1, :, :]  # shape: (batch_size, output_size)
val_acc = compute_accuracy(final_test_output, tstY)
print(f" RNN Test Set Accuracy: {val_acc:.2f}")

y_pred = np.argmax(final_test_output, axis=1)
y_true = np.argmax(tstY, axis=1)

class_names = ["1","2","3","4","5","6"]
title= "RNN Confusion Matrix for test set"
cm_rnn = plot_confusion_matrix(y_true, y_pred, class_names,title)

# Forward Pass (over all timesteps)
output_train, _ = RNN_forward_t_times(parameters, trX)

# output_list: (time_steps, batch_size, output_size)
# We assume loss at the final timestep only
final_train_output = output_train[-1, :, :]  # shape: (batch_size, output_size)
val_acc = compute_accuracy(final_train_output, trY)

print(f'Train Accuracy RNN {val_acc}')

y_pred = np.argmax(final_train_output, axis=1)
y_true = np.argmax(trY, axis=1)

class_names = ["1","2","3","4","5","6"]
```

```
title= "RNN Confusion Matrix for training set"
cm_rnn = plot_confusion_matrix(y_true, y_pred, class_names,title)


# Part B

def LSTM_weights(input_size,hidden_size,output_size):

    np.random.seed(38)

    # Xavier Uniform Initialization for weights
    limit_Wf = np.sqrt(6 / (input_size  + hidden_size+ hidden_size))
    limit_Wo = np.sqrt(6 / (hidden_size + output_size))


    # init weights for forget gate
    Wf = np.random.uniform(-limit_Wf, limit_Wf, (input_size + hidden_size, hidden_size))
    bf =  np.random.uniform(-limit_Wf, limit_Wf,(1, hidden_size))

    # init weights for input gate
    Wi = np.random.uniform(-limit_Wf, limit_Wf, (input_size + hidden_size, hidden_size))
    bi = np.random.uniform(-limit_Wf, limit_Wf, (1, hidden_size))


    Wc = np.random.uniform(-limit_Wf, limit_Wf,(input_size + hidden_size, hidden_size))
    bc = np.random.uniform(-limit_Wf, limit_Wf, (1, hidden_size))

    # init weigths output gate
    Wo = np.random.uniform(-limit_Wf, limit_Wf, (input_size + hidden_size, hidden_size))
    bo = np.random.uniform(-limit_Wf, limit_Wf, (1, hidden_size))

    # Dense Hidden layers
    Whd = np.random.uniform(-limit_Wo, limit_Wo, (hidden_size, output_size))
    bhd = np.random.uniform(-limit_Wo, limit_Wo, (1, output_size))


    parameters = {
        "Wf": Wf,
        "bf": bf,
        "Wi": Wi,
        "bi": bi,
        "Wc": Wc,
        "bc": bc,
        "Wo": Wo,
        "bo":bo,
        "Whd":Whd,
        "bhd":bhd
    }
    return parameters

# Initiliaze Momentum weigths
def init_momentum_weight_lstm(input_size,hidden_size,output_size):

    dWf = np.zeros((input_size + hidden_size, hidden_size))
    dbf = np.zeros((1, hidden_size))

    dWi = np.zeros((input_size + hidden_size, hidden_size))
    dbi = np.zeros((1, hidden_size))

    dWc = np.zeros((input_size + hidden_size, hidden_size))
    dbc = np.zeros((1, hidden_size))
```

```
    dWo = np.zeros((input_size + hidden_size, hidden_size))
    dbo = np.zeros((1, hidden_size))

    dWhd = np.zeros((hidden_size, output_size))
    dbhd = np.zeros((1, output_size))


    momentum = {
        "dWf": dWf,
        "dbf": dbf,
        "dWi": dWi,
        "dbi": dbi,
        "dWc": dWc,
        "dbc": dbc,
        "dWo": dWo,
        "dbo": dbo,
        "dWhd":dWhd,
        "dbhd":dbhd
    }
    return momentum

def LSTM_forward(parameters,ht_prev,xt,ct_prev):

    # Retrieve parameters from "parameters"
    Wf = parameters["Wf"]
    bf = parameters["bf"]
    Wi = parameters["Wi"]
    bi = parameters["bi"]
    Wc = parameters["Wc"]
    bc = parameters["bc"]
    Wo = parameters["Wo"]
    bo = parameters["bo"]


    concat = np.concatenate([ht_prev, xt], axis=1)

    # Forget gate
    forget_gate = sigmoid(concat @ Wf + bf)

    # Input Gate
    input_gate = sigmoid(concat @ Wi  + bi)
    c_t_prime = np.tanh(concat @  Wc + bc)

    # Memory Update
    ct = forget_gate * ct_prev + input_gate * c_t_prime

    # Output Gate
    output_gate = sigmoid(concat @ Wo + bo)
    ht =  np.tanh(ct) * output_gate

    output_param_LSTM = {
        "forget gate": forget_gate,
        "input gate": input_gate,
        "c_t_prime":c_t_prime,
        "output gate":output_gate,
        "ct":ct,
        "ht":ht
    }

    return output_param_LSTM

def dense_lstm(parameters,input_dense):
```

```
    Whd = parameters["Whd"]
    bhd = parameters["bhd"]

    temp_dense =  input_dense @ Whd  + bhd
    output_dense = softmax(temp_dense)

    return output_dense

def LSTM_forward_t_times(parameters,x,hidden_size):

    n_samples, time, n_features = x.shape

    # Create variables
    ht_prev = np.zeros((n_samples,hidden_size))
    ct_prev = np.zeros_like(ht_prev)

    forget_gate_list = []
    input_gate_list  = []
    c_t_prime_list = []
    output_gate_list  = []
    ct_list = []
    hidden_list = []

    for t in range(time):
        cache = LSTM_forward(parameters,ht_prev,x[:,t,:],ct_prev)


        output_gate_list.append(cache["output gate"])
        forget_gate_list.append(cache["forget gate"])
        input_gate_list.append(cache["input gate"])
        c_t_prime_list.append(cache["c_t_prime"])
        hidden_list.append(cache["ht"])
        ct_list.append(cache["ct"])


        ct_prev = cache["ct"]
        ht_prev = cache["ht"]



    output_lstm = dense_lstm(parameters,hidden_list[-1])

    return np.array(hidden_list), np.array(ct_list), np.array(forget_gate_list), np.array(c_t_prime_list),
np.array(output_gate_list),np.array(input_gate_list), output_lstm

# BPTT for LSTM
def bptt_lstm(parameters, ht, xt, ct, ft, ct_l, ot,it, y_pred, y_true):
    """
    Perform Backpropagation Through Time for LSTM.

    Args:
        parameters (dict): Dictionary containing LSTM parameters.
        ht (np.ndarray): Hidden states, shape (time, n_samples, hidden_size).
        xt (np.ndarray): Input sequences, shape (n_samples, time, n_features).
        ct (np.ndarray): Cell states, shape (time, n_samples, hidden_size).
        ft (np.ndarray): Forget gate activations, shape (time, n_samples, hidden_size).
        ct_l (np.ndarray): Cell state after activation, shape (time, n_samples, hidden_size).
        ot (np.ndarray): Output gate activations, shape (time, n_samples, hidden_size).
        y_pred (np.ndarray): Predictions, shape (n_samples, output_size).
        y_true (np.ndarray): True labels, shape (n_samples, output_size).
        hidden_list (list or np.ndarray): List of hidden states for each time step.
```

```
Returns:
    grads (dict): Gradients of all parameters.
"""

n_samples, time, n_features = xt.shape

hidden_size = ht.shape[2]
output_size = y_pred.shape[1]


# Unpack parameters
Wf = parameters["Wf"]  # Shape: (hidden_size, hidden_size + n_features)
bf = parameters["bf"]  # Shape: (1, hidden_size)
Wi = parameters["Wi"]
bi = parameters["bi"]
Wc = parameters["Wc"]
bc = parameters["bc"]
Wo = parameters["Wo"]
bo = parameters["bo"]
Whd = parameters["Whd"]  # Output layer weights
bhd = parameters["bhd"]    # Output layer bias

# Initialize gradients
dWf, dbf = np.zeros_like(Wf), np.zeros_like(bf)
dWi, dbi = np.zeros_like(Wi), np.zeros_like(bi)
dWc, dbc = np.zeros_like(Wc), np.zeros_like(bc)
dWo, dbo = np.zeros_like(Wo), np.zeros_like(bo)
dWhd, dbhd = np.zeros_like(Whd), np.zeros_like(bhd)

# Compute gradient for the output layer
dZo = y_pred - y_true  # Shape: (n_samples, output_size)


# Assuming Whd maps from hidden_size to output_size
final_h = ht[-1]  # Shape: (n_samples, hidden_size)
dWhd = np.dot(final_h.T, dZo)  # Shape: (hidden_size, output_size)
dbhd = np.sum(dZo, axis=0, keepdims=True)  # Shape: (1, output_size)

# Initialize gradient for the hidden state
d_next = np.dot(dZo, Whd.T)  # Shape: (n_samples, hidden_size)

# Backpropagate through time
dht_next = np.zeros((n_samples, ht.shape[2]))
dct_next = np.zeros_like(dht_next)


for t in reversed(range(time)):
    # Current gate and cell states
    ct_temp = ct[t]  # Shape: (n_samples, hidden_size)
    ft_temp = ft[t]
    it_temp = it[t]
    ot_temp = ot[t]
    ct_l_temp = ct_l[t]

    # Previous hidden state
    if t == 0:
        ht_prev = np.zeros((n_samples, hidden_size))
        ct_prev = np.zeros((n_samples, hidden_size))
    else:
        ht_prev = ht[t - 1]
        ct_prev = ct[t - 1]
```

```
        concat = np.concatenate([ht_prev, xt[:, t, :]], axis=1)

        dht = d_next + dht_next  # Aggregate gradients from the future
        do = dht * np.tanh(ct_temp)  # Derivative of output gate
        do_sigmoid = do * ot_temp * (1 - ot_temp)

        dWo += np.dot(concat.T, do_sigmoid)
        dbo += np.sum(do_sigmoid, axis=0, keepdims=True)

        dct = dht * ot_temp * (1 - np.tanh(ct_temp)**2) + dct_next

        df = dct * ct_prev # Gradient of forget gate
        df_sigmoid = df * ft_temp * (1 - ft_temp)
        dWf += np.dot(concat.T, df_sigmoid)
        dbf += np.sum(df_sigmoid, axis=0, keepdims=True)

        di = dct * ct_l_temp  # Input gate gradient
        di_sigmoid = di * it_temp * (1 - it_temp)

        dWi += np.dot(concat.T, di_sigmoid)
        dbi += np.sum(di_sigmoid, axis=0, keepdims=True)

        dc_bar = dct * it_temp  # Candidate gradient
        dc_bar_tanh = dc_bar * (1 - ct_l_temp**2)

        dWc += np.dot(concat.T, dc_bar_tanh)
        dbc += np.sum(dc_bar_tanh, axis=0, keepdims=True)

        # Backpropagate to the previous time step
        d_prev_concat = (
            np.dot(df_sigmoid, Wf.T) +
            np.dot(di_sigmoid, Wi.T) +
            np.dot(dc_bar_tanh, Wc.T) +
            np.dot(do_sigmoid, Wo.T)
        )

        dht_next = d_prev_concat[:, :ht.shape[2]]
        dct_next = dct * ft_temp  # Carry the gradient of cell state
    # Aggregate gradients
    grads = {
        "dWf": dWf,
        "dbf": dbf,
        "dWi": dWi,
        "dbi": dbi,
        "dWc": dWc,
        "dbc": dbc,
        "dWo": dWo,
        "dbo": dbo,
        "dWhd": dWhd,
        "dbhd": dbhd
    }

    return grads

# Update Parameters for LSTM

def update_parameters_lstm(parameters, momentum, grads, alpha, learning_rate):
    # alpha is momentum factor
    # learning_rate is step size
    for param_name in ["Wf","bf","Wi","bi","Wc","bc","Wo","bo","Whd","bhd"]:
        dparam_name = "d" + param_name
        momentum[dparam_name] = alpha * momentum[dparam_name] - learning_rate * grads[dparam_name]
```

```
        parameters[param_name] += momentum[dparam_name]

    return parameters, momentum


# Initalize Parameters
alpha = 0.75
learning_rate = 0.0002
mini_batch = 32
epoch_size = 50

patience = 3
best_val_cost = 999
epoch_interpt = 0

total_time = 0

# Store values

train_cost_list_lstm = []
val_cost_list_lstm = []
train_acc_list_lstm = []
val_acc_list_lstm = []


parameters_lstm = LSTM_weights(input_size=3,hidden_size=64,output_size=6)
momentum= init_momentum_weight_lstm(input_size=3,hidden_size=64,output_size=6)

for epoch in range(epoch_size):

    start_time = time.time()
    np.random.seed(38)

    # Shuffle dataset
    indices = np.arange(trX.shape[0])
    np.random.shuffle(indices)
    trx_shuffle_x = trX[indices, :, :]
    try_shuffle_y = trY[indices, :]

    # Split into train/val
    length_tr = trX.shape[0]
    train_set_length = int(length_tr * 0.8)

    train_X = trx_shuffle_x[:train_set_length,:,:]
    train_Y = try_shuffle_y[:train_set_length,:]

    val_X = trx_shuffle_x[train_set_length:,:,:]
    val_Y = try_shuffle_y[train_set_length:,:]

    start = 0
    end = mini_batch
    temp_cost = 0

    num_batches = train_X.shape[0] // mini_batch
    train_acc = 0
    for j in range(num_batches):
        batch_x = train_X[start:end,:,:]
        batch_y = train_Y[start:end,:]

        # Forward Pass (over all timesteps)
        hidden_list,ct_list, forget_gate_list, c_t_prime_list, output_gate_list,input_gate_list, output_lstm =
LSTM_forward_t_times(parameters_lstm, batch_x,hidden_size=64)
```

```
            # Calculate cost
            cost = cost_rnn(batch_y, output_lstm)
            train_acc_temp = compute_accuracy(output_lstm, batch_y)
            train_acc += train_acc_temp
            temp_cost += cost

            # Get gradients
            grads = bptt_lstm(parameters_lstm, ht=hidden_list, xt=batch_x, ct=ct_list, ft=forget_gate_list, ct_l=c_t_prime_list,
ot=output_gate_list,it=input_gate_list, y_pred=output_lstm, y_true=batch_y)
            for grad in grads.values():
                np.clip(grad, -5, 5, out=grad)

            # Update Parameters
            parameters_lstm, momentum = update_parameters_lstm(parameters_lstm, momentum, grads, alpha, learning_rate)

            start = end
            end += mini_batch

        # Validation
        _,_, _, _, _,_,val_out= LSTM_forward_t_times(parameters_lstm, val_X,hidden_size=64)
        val_cost = cost_rnn(val_Y, val_out)
        val_acc = compute_accuracy(val_out, val_Y)

        total_cost_train = temp_cost / num_batches
        train_acc /= num_batches

        end_time = time.time()
        elapsed_time = end_time - start_time

        print(f'Epoch {epoch +1 }: Train Acc: {train_acc:.2f} % Train Cost: {total_cost_train:.3f},  '
            f'Val Cost: {val_cost:.3f}, Val Acc: {val_acc:.2f}%, '
            f'Time: {int(elapsed_time)}s')

        # Saving Parameters
        train_cost_list_lstm.append(total_cost_train)

        val_cost_list_lstm.append(val_cost)

        train_acc_list_lstm.append(train_acc)

        val_acc_list_lstm.append(val_acc)

        # Early stopping
        if val_cost < best_val_cost:
            best_val_cost = val_cost
            epoch_interpt = 0
        else:
            epoch_interpt += 1
            if epoch_interpt >= patience:

                print("Early Stopped!")
                break


    # Plot the plots

    # Forward Pass (over all timesteps)
    _,_, _, _, _,_,final_test_output = LSTM_forward_t_times(parameters_lstm, tstX,hidden_size=64)

    # output_list: (time_steps, batch_size, output_size)
```

```
# We assume loss at the final timestep only

val_acc = compute_accuracy(final_test_output, tstY)
print(f'Test Accuracy LSTM: {val_acc}')

y_pred = np.argmax(final_test_output, axis=1)
y_true = np.argmax(tstY, axis=1)

class_names = ["1","2","3","4","5","6"]
title= "LSTM Confusion Matrix for Test Data Set "
cm_lstm = plot_confusion_matrix(y_true, y_pred, class_names,title)

# Forward Pass (over all timesteps)
_,_, _, _, _,_,final_train_output = LSTM_forward_t_times(parameters_lstm, trX,hidden_size=64)

# output_list: (time_steps, batch_size, output_size)
# We assume loss at the final timestep only
val_acc = compute_accuracy(final_train_output, trY)

print(f'Train Accuracy LSTM: {val_acc}')

y_pred = np.argmax(final_train_output, axis=1)
y_true = np.argmax(trY, axis=1)

class_names = ["1","2","3","4","5","6"]
title= "LSTM Confusion Matrix for training set"
cm_lstm = plot_confusion_matrix(y_true, y_pred, class_names,title)

# Create subplots
fig, axs = plt.subplots(2, 2, figsize=(20, 10))

# Train loss plot
axs[0, 0].plot(train_cost_list_lstm, color='green')
axs[0, 0].set_title('Train Cross Entropy Loss')
axs[0, 0].set_xlabel('Epoch')
axs[0, 0].set_ylabel('Loss')

# Validation loss plot
axs[0, 1].plot(val_cost_list_lstm, color='red')
axs[0, 1].set_title('Validation Cross Entropy Loss')
axs[0, 1].set_xlabel('Epoch')
axs[0, 1].set_ylabel('Loss')

# Train accuracy plot
axs[1, 0].plot(train_acc_list_lstm, color='green')
axs[1, 0].set_title('Train Accuracy')
axs[1, 0].set_xlabel('Epoch')
axs[1, 0].set_ylabel('Accuracy')

# Validation accuracy plot
axs[1, 1].plot(val_acc_list_lstm, color='red')
axs[1, 1].set_title('Validation Accuracy')
axs[1, 1].set_xlabel('Epoch')
axs[1, 1].set_ylabel('Accuracy')

# Add an overall title
fig.suptitle(
    "LSTM \nLearning Rate = 0.0002 | Momentum = 0.75 | Batch Size = 32 | Hidden Neuron = 64 | Patience = 3  ",
    fontsize=12
)

plt.show()
```

Part C

```
def init_gru_weights(input_size,hidden_size,output_size):

    limit_xh = np.sqrt(6 / (input_size + hidden_size))
    limit_hh = np.sqrt(6 / (hidden_size + hidden_size))
    limit = np.sqrt(6 / (output_size + hidden_size))


    # Update Gate parameters
    Wz = np.random.uniform(-limit_xh, limit_xh, (input_size, hidden_size))
    Uz = np.random.uniform(-limit_hh, limit_hh, (hidden_size, hidden_size))
    bz = np.zeros((1, hidden_size))

    # Reset Gate parameters
    Wr = np.random.uniform(-limit_xh, limit_xh, (input_size, hidden_size))
    Ur = np.random.uniform(-limit_hh, limit_hh, (hidden_size, hidden_size))
    br = np.zeros((1, hidden_size))

    # Candidate Hidden State parameters
    Wh = np.random.uniform(-limit_xh, limit_xh, (input_size, hidden_size))
    Uh = np.random.uniform(-limit_hh, limit_hh, (hidden_size, hidden_size))
    bh = np.zeros((1, hidden_size))

    # Output Layer parameters
    Wy = np.random.uniform(-limit, limit, (hidden_size, output_size))
    by = np.zeros((1, output_size))

    parameters = {
        "Wz": Wz, "Uz": Uz, "bz": bz,
        "Wr": Wr, "Ur": Ur, "br": br,
        "Wh": Wh, "Uh": Uh, "bh": bh,
        "Wy": Wy, "by": by
    }

    return parameters

def init_gru_momentum(input_size,hidden_size,output_size):

    # Update Gate parameters
    dWz = np.zeros((input_size, hidden_size))
    dUz = np.zeros((hidden_size, hidden_size))
    dbz = np.zeros((1, hidden_size))

    # Reset Gate parameters
    dWr = np.zeros((input_size, hidden_size))
    dUr = np.zeros((hidden_size, hidden_size))
    dbr = np.zeros((1, hidden_size))

    # Candidate Hidden State parameters
    dWh = np.zeros((input_size, hidden_size))
    dUh = np.zeros((hidden_size, hidden_size))
    dbh = np.zeros((1, hidden_size))

    # Output Layer parameters
    dWy = np.zeros((hidden_size, output_size))
    dby = np.zeros((1, output_size))

    momentum = {
        "dWz": dWz, "dUz": dUz, "dbz": dbz,
        "dWr": dWr, "dUr": dUr, "dbr": dbr,
```

```
        "dWh": dWh, "dUh": dUh, "dbh": dbh,
        "dWy": dWy, "dby": dby
    }

    return momentum

def sigmoid(X):
    # Clip X to the range [-709, 709] to prevent overflow in exp
    X_clipped = np.clip(X, -709, 709)
    return 1 / (1 + np.exp(-X_clipped))

def forward_pass_gru(parameters,ht_prev,xt):

    Wz =parameters["Wz"]
    Uz =parameters["Uz"]
    bz =parameters["bz"]

    Wr =parameters["Wr"]
    Ur =parameters["Ur"]
    br =parameters["br"]

    Wh =parameters["Wh"]
    Uh =parameters["Uh"]
    bh =parameters["bh"]


    zt = sigmoid(np.dot(xt,Wz) + np.dot(ht_prev,Uz) + bz)

    rt = sigmoid(np.dot(xt,Wr) + np.dot(ht_prev,Ur) + br)

    ht_l = np.tanh(np.dot(xt,Wh) + np.dot((rt * ht_prev),Uh) + bh)
    ht  = np.multiply(zt,ht_prev) + np.multiply((1-zt),ht_l)

    return ht,zt,rt,ht_l

def forward_pass_gru_times(parameters,x,hidden_size):
    n_samples, time, n_features = x.shape

    # Initalize values
    ht_prev = np.zeros((n_samples,hidden_size))
    Wy = parameters["Wy"]
    by = parameters["by"]

    hidden_list = []
    z_list = []
    r_list  = []
    ht_l_list = []

    for t in range(time):

        h_out,zt,rt,ht_l_out = forward_pass_gru(parameters,ht_prev,x[:,t,:])
        ht_prev  = h_out
        hidden_list.append(ht_prev)
        z_list.append(zt)
        r_list.append(rt)
        ht_l_list.append(ht_l_out)

    temp = np.dot(h_out,Wy) + by
    output = softmax(temp)

    return output, np.array(hidden_list),np.array(z_list),np.array(r_list),np.array(ht_l_list)
```

```
def bptt_gru(parameters,x,y_pred,hidden_list,y_true,zt,ht_l,rt):

    n_samples, time, n_features = x.shape

    # Parameters
    Wz =parameters["Wz"]
    Uz =parameters["Uz"]
    bz =parameters["bz"]

    Wr =parameters["Wr"]
    Ur =parameters["Ur"]
    br =parameters["br"]

    Wh =parameters["Wh"]
    Uh =parameters["Uh"]
    bh =parameters["bh"]

    Wy = parameters["Wy"]


    # Initialize derivatives
    dWz, dUz, dbz = np.zeros_like(Wz),np.zeros_like(Uz),np.zeros_like(bz)
    dWr, dUr, dbr = np.zeros_like(Wr),np.zeros_like(Ur),np.zeros_like(br)
    dWh, dUh, dbh = np.zeros_like(Wh),np.zeros_like(Uh),np.zeros_like(bh)

    # Update in output layer
    dZo = y_pred - y_true  # shape: (n_samples, output_size)

    # Use the final hidden state: hidden[-1,:,:] shape (n_samples, hidden_size)
    final_h = hidden_list[-1, :, :]
    dWy = np.dot(final_h.T, dZo)
    dby = np.sum(dZo, axis=0, keepdims=True)

    # Backprop through time
    dhnext = np.zeros((n_samples, Ur.shape[0]))

    for t in reversed(range(time)):

        zt_temp = zt[t]
        ht_l_temp = ht_l[t]
        rt_temp =rt[t]

        if t == 0:
            h_prev = np.zeros((n_samples, Ur.shape[0]))
        else:
            h_prev = hidden_list[t-1]

        if (t == time-1 ):
            dht = np.dot(dZo, Wy.T) + dhnext  # Only the final time step receives dZo
        else:
            dht = dhnext


        dht_l = dht*(1- zt_temp)
        dht_l2 = dht_l*(1-(ht_l_temp**2))

        dWh += np.dot(x[:,t,:].T,dht_l2)
        assert(dWh.shape == Wh.shape)
        temp_u = rt_temp*h_prev
        dUh += np.dot(dht_l2.T,temp_u).T
        assert(dUh.shape == Uh.shape)
        dbh +=  np.sum(dht_l2,axis=0,keepdims=True)
```

```
        assert(dbh.shape == bh.shape)

        drt = dht_l2*np.dot(h_prev,Uh)
        drt2 = drt*rt[t]*(1-rt_temp)

        dWr += np.dot(x[:,t,:].T,drt2)
        dUr += np.dot(drt2.T,h_prev).T
        dbr += np.sum(drt2,axis=0,keepdims=True)

        dzt = dht*(h_prev - ht_l_temp)
        dzt2 = dzt*zt_temp*(1-zt_temp)

        dWz += np.dot(x[:,t,:].T,dzt2)
        dUz += np.dot(dzt2.T,h_prev).T
        dbz += np.sum(dzt2,axis=0,keepdims=True)


        dhnext = (dht * zt_temp) + (dht_l2 @ Uh.T) * (1 - zt_temp) + (dzt2 @ Uz.T)

    # Check all


    grads = {
        "dWz": dWz, "dUz": dUz, "dbz": dbz,
        "dWr": dWr, "dUr": dUr, "dbr": dbr,
        "dWh": dWh, "dUh": dUh, "dbh": dbh,
        "dWy": dWy, "dby": dby

    }

    return grads

def update_parameters_gru(parameters, momentum, grads, alpha, learning_rate):
    # alpha is momentum factor
    # learning_rate is step size
    for param_name in ["Wz","Uz","bz","Wr", "Ur","br","Wh","Uh","bh","Wy","by"]:
        dparam_name = "d" + param_name
        momentum[dparam_name] = alpha * momentum[dparam_name] - learning_rate * grads[dparam_name]
        parameters[param_name] += momentum[dparam_name]

    return parameters, momentum


def cross_entropy(target,pred):
    return -np.mean(np.sum(target * np.log(pred + 1e-9), axis=1))

# Train the model



# initliaze weights
parameters2 =init_gru_weights(3,32,6)
momentum = init_gru_momentum(3,32,6)

# Train the model

# Initalize Parameters
alpha = 0.85
learning_rate = 0.005
mini_batch = 32
epoch_size = 50
```

```
patience = 3

best_val_cost = 999
epoch_interpt = 0

total_time = 0

# Store values
train_cost_list_gru = []
val_cost_list_gru = []
train_acc_list_gru = []
val_acc_list_gru = []

for epoch in range(epoch_size):

    start_time = time.time()
    np.random.seed(38)

    # Shuffle dataset
    indices = np.arange(trX.shape[0])
    np.random.shuffle(indices)
    trx_shuffle_x = trX[indices, :, :]
    try_shuffle_y = trY[indices, :]

    # Split into train/val
    length_tr = trX.shape[0]
    train_set_length = int(length_tr * 0.9)

    train_X = trx_shuffle_x[:train_set_length,:,:]
    train_Y = try_shuffle_y[:train_set_length,:]

    val_X = trx_shuffle_x[train_set_length:,:,:]
    val_Y = try_shuffle_y[train_set_length:,:]

    start = 0
    end = mini_batch
    temp_cost = 0
    train_acc = 0

    num_batches = train_X.shape[0] // mini_batch



    for j in range(num_batches):
        batch_x = train_X[start:end]
        batch_y = train_Y[start:end]

        # Forward Pass (over all timesteps)
        output,hidden_list,z_list,r_list,ht_l_list= forward_pass_gru_times(parameters=parameters2,x=batch_x,hidden_size=32)


        # Calculate cost
        cost = cross_entropy(batch_y, output)
        temp_cost += cost

        train_acc_temp = compute_accuracy(output, batch_y)
        train_acc += train_acc_temp


        # Get gradients
        grads =  bptt_gru(parameters2,batch_x,output,hidden_list,batch_y,z_list,ht_l_list,r_list)
```

```
        # Update Parameters
        parameters2, momentum = update_parameters_gru(parameters2, momentum, grads, alpha, learning_rate)

        start = end
        end += mini_batch

    # Validation
    val_out,_,_,_,_ = forward_pass_gru_times(parameters2, val_X,hidden_size=32)
    val_cost = cross_entropy(val_Y, val_out)
    val_acc = compute_accuracy(val_out, val_Y)

    total_cost_train = temp_cost / num_batches
    train_acc /= num_batches

    end_time = time.time()
    elapsed_time = end_time - start_time

    print(f'Epoch {epoch +1 }: Train Cost: {total_cost_train:.3f} Train Acc {train_acc:.2f} % , '
        f'Val Cost: {val_cost:.3f}, Val Acc: {val_acc:.2f}%, '
        f'Time: {int(elapsed_time)}s')

    train_cost_list_gru.append(total_cost_train)

    val_cost_list_gru.append(val_cost)

    train_acc_list_gru.append(train_acc)

    val_acc_list_gru.append(val_acc)

    # Early stopping
    if val_cost < best_val_cost:
        best_val_cost = val_cost
        epoch_interpt = 0
    else:
        epoch_interpt += 1
        if epoch_interpt >= patience:
            print("Early Stopped!")
            break

# Forward Pass (over all timesteps)
output_test, _,_,_,_ = forward_pass_gru_times(parameters2, tstX,hidden_size=32)

# output_list: (time_steps, batch_size, output_size)
# We assume loss at the final timestep only

val_acc = compute_accuracy(output_test, tstY)
print(f'Test Accuracy Gru {val_acc}')

y_pred = np.argmax(output_test, axis=1)
y_true = np.argmax(tstY, axis=1)

class_names = ["1","2","3","4","5","6"]
title= "GRU Confusion Matrix for test"
cm_gru = plot_confusion_matrix(y_true, y_pred, class_names,title)

# Forward Pass (over all timesteps)
output_train, _,_,_,_ = forward_pass_gru_times(parameters2, trX,hidden_size=32)

# output_list: (time_steps, batch_size, output_size)
# We assume loss at the final timestep only

val_acc = compute_accuracy(output_train, trY)
```

```
        print(f'Train Accuracy Gru {val_acc}')

        y_pred = np.argmax(output_train, axis=1)
        y_true = np.argmax(trY, axis=1)

        class_names = ["1","2","3","4","5","6"]
        title= "GRU Confusion Matrix for train"
        cm_gru = plot_confusion_matrix(y_true, y_pred, class_names,title)

        # Create subplots
        fig, axs = plt.subplots(2, 2, figsize=(20, 10))

        # Train loss plot
        axs[0, 0].plot(train_cost_list_gru, color='green')
        axs[0, 0].set_title('Train Cross Entropy Loss')
        axs[0, 0].set_xlabel('Epoch')
        axs[0, 0].set_ylabel('Loss')

        # Validation loss plot
        axs[0, 1].plot(val_cost_list_gru, color='red')
        axs[0, 1].set_title('Validation Cross Entropy Loss')
        axs[0, 1].set_xlabel('Epoch')
        axs[0, 1].set_ylabel('Loss')

        # Train accuracy plot
        axs[1, 0].plot(train_acc_list_gru, color='green')
        axs[1, 0].set_title('Train Accuracy')
        axs[1, 0].set_xlabel('Epoch')
        axs[1, 0].set_ylabel('Accuracy')

        # Validation accuracy plot
        axs[1, 1].plot(val_acc_list_gru, color='red')
        axs[1, 1].set_title('Validation Accuracy')
        axs[1, 1].set_xlabel('Epoch')
        axs[1, 1].set_ylabel('Accuracy')

        # Add an overall title
        fig.suptitle(
            "GRU \nLearning Rate = 0.005 | Momentum = 0.85 | Batch Size = 32 | Hidden Neuron = 32 | Patience = 3  ",
            fontsize=12
        )


        plt.show()
    else:
        print("Invalid question number. Please select 1, 2, or 3.")




# Main execution block
if __name__ == "__main__":
    # Ensure a command-line argument is passed
    if len(sys.argv) > 1:
        question = sys.argv[1]
        MustafaCankan_balci_22101761_hw1(question)
    else:
        print("Please provide a question number as a command-line argument.")
```