

EEE 443/543: Neural Networks

Mini Project Report

Name: Ozan Cem Bař

ID: 22102757

Date: December 22, 2024

December 22, 2024

Question-1.

1.a) Preprocessing The Images



Figure 1: 200 randomly chosen colored images.

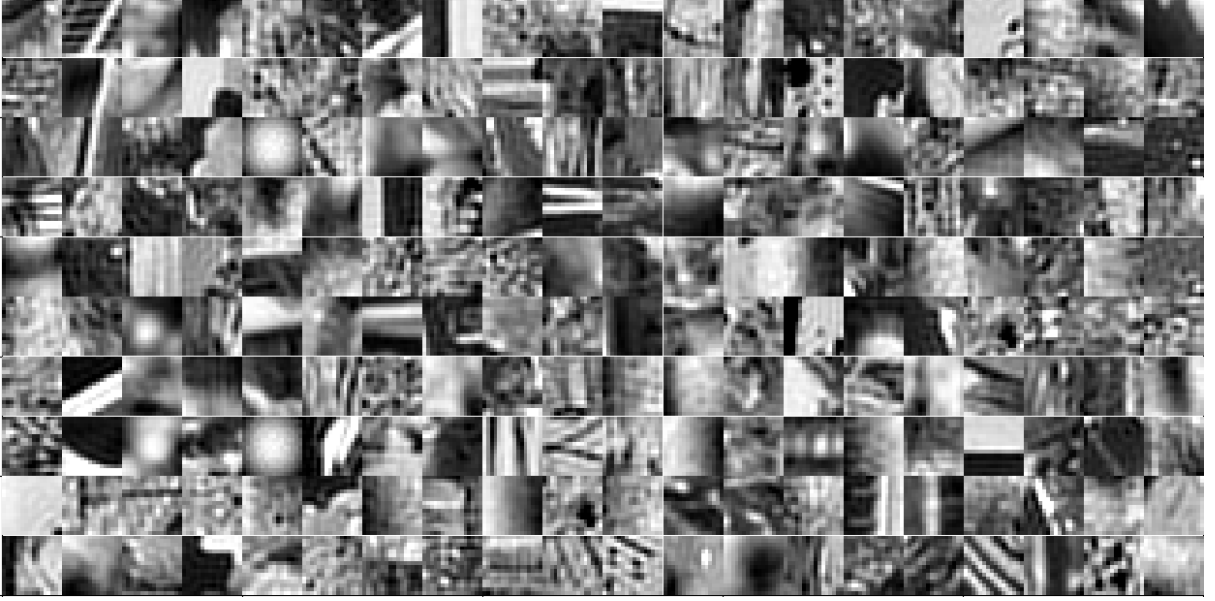


Figure 2: 200 randomly chosen images after converting to gray-scale and normalizing.

The images in the data set are processed and normalized as instructed. In figure1, the actual images are seen, and in figure2, the processed images are seen. The luminosity value $Y = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B$ that is used to convert the images to gray-scale allows me to see color gradients that were otherwise not visible to the eye. Even in smooth-looking images, after converting them to gray-scale following the instructions, some edges and gradients in the colors become visible.

1.b) Code Structure and Training Optimization

The instructions for initializing the auto-encoder and training the network are followed. Here are the steps I took and how I implemented them in my code:

- A function named *init_AE_Wb* was written that returns the network weights $W^{(1)}$ and $W^{(2)}$ and biases $b^{(1)}$ and $b^{(2)}$ by randomly sampling their elements from a uniform distribution from the interval $[-w_0, w_0]$. Here,

$$w_0 = \sqrt{\frac{6}{L_{pre} + L_{post}}} \quad (1)$$

where $L_{pre,post}$ are the input and output dimensions of the weight matrices.

- A cost function *aeCost* was written that takes a dictionary of weights and biases as input and calculates the cost value and its gradients with respect to the parameters. With the added terms in the cost function, the gradients of the parameters are calculated as follows:

$$\begin{aligned}
\frac{\partial J}{\partial W^{(2)}} &= \frac{1}{N} \left(\frac{\partial J}{\partial Z^{(2)}} \cdot A^{(1)T} \right) + \lambda \cdot W^{(2)} \\
\frac{\partial J}{\partial b^{(2)}} &= \frac{1}{N} \sum \frac{\partial J}{\partial Z^{(2)}} \\
\frac{\partial J}{\partial W^{(1)}} &= \frac{1}{N} \left(\frac{\partial J}{\partial Z^{(1)}} \cdot A^{(0)T} \right) + \lambda \cdot W^{(1)} \\
\frac{\partial J}{\partial b^{(1)}} &= \frac{1}{N} \sum \frac{\partial J}{\partial Z^{(1)}}
\end{aligned}$$

- Sigmoid function was used as the activation function for the hidden layer and the output layer.
- The Kullback-Leiber divergence term in the loss modifies the $\frac{\partial J}{\partial A^{(1)}}$ by introducing and addition of the gradient of Kullback-Leiber divergence with respect to hidden activations $A^{(1)}$.
- Finally, *solver* function is implemented that takes in the weights, biases, and the data to be trained against, and optimizes the network parameters by implementing Adam optimizer on the gradient values returned by the *aeCost* function.

These steps are followed for $L_{hid} = 64$ and $\lambda = 5 \cdot 10^{-4}$, and after experimenting with the values of β and ρ , I found the optimal loss function and training parameters as:

$$\begin{aligned}
\beta &= 0.1 \\
\rho &= 0.5 \\
\text{Batch Size} &= 32 \\
\text{epochs} &= 20
\end{aligned}$$

Please note that the parameters of the Adam optimizer are taken as $\beta_1 = 0.9$, $\beta_2 = 0.99$ and these values do not have a correlation to β in the loss function.

1.c) Hidden Layer Weights



Figure 3: Connection weights of the trained hidden neurons for $L_{hid} = 64$

In the figure 3, the connection weights for each neuron in the hidden layer can be seen. The bright areas for each neurons' connection weights display how much their activation depends on the different parts of the image. It can be seen that some neurons are more susceptible to different parts of the image. Some neurons are seen to be more sensitive to circular shapes in the images while some are seen to be more sensitive to dot like shapes or rod-like long bright area. However, the connection weights themselves are not similar, or representative of the image patches that the auto-encoder was initially trained on.

1.d) Effects of Changing Network Parameters

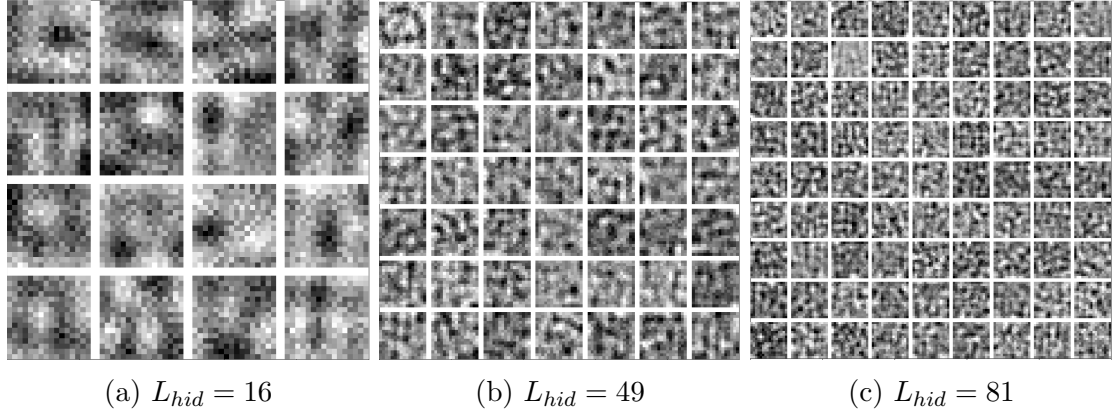


Figure 4: Connection weights of each neuron for $\lambda = 10^{-5}$

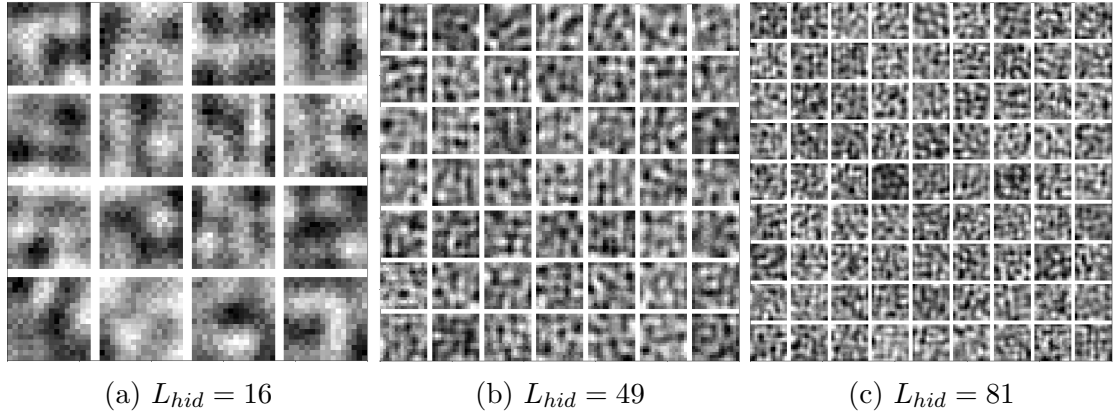


Figure 5: Connection weights of each neuron for $\lambda = 10^{-4}$

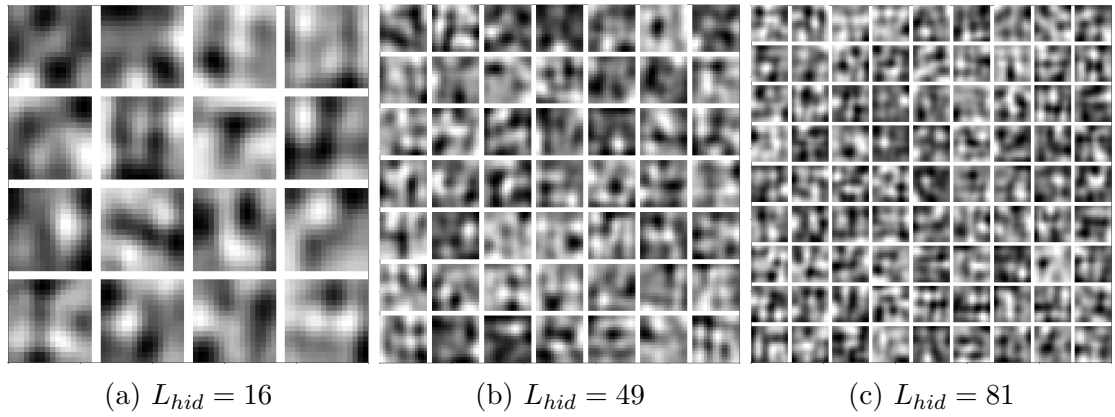


Figure 6: Connection weights of each neuron for $\lambda = 10^{-3}$

By choosing different values for the cost coefficient of weights $\lambda = [10^{-5}, 10^{-4}, 10^{-3}]$ and the hidden layer size $L_{hid} = [16, 49, 81]$, the auto-encoder was tested for 9 different values.

The affect of choosing different values on the hidden layer neurons' connection weights can be observed in figures 4, 5, and 6. Here are some of my findings:

- What is common for all three λ values is that as hidden layer size increases, the connection weights of each neuron learn more complicated patterns and shapes.
- Increasing the λ value seems to smooth out the connection weights and make features that are extracted clearer.
- Decreasing the λ value below 10^{-4} introduces noise to the connection weights and prevents extracting high quality features.
- High L_{hid} values together with low λ values results with almost random looking features as seen in figure 4c.
- From my personal experience, increasing λ below 10^{-3} causes the extracted features to be too general, that do not perform well.

Question 2.

2.a) Code Structure and Affect of Network Parameters

In this question, a two layer neural network with an added embedding layer to predict fourth word that comes after the three-word sequence data. Here is the flow of processes I followed:

- The function *init_NLP_Wb* is called to initialize the network weights and biases by random sampling from a Gaussian distribution with standard variation of 0.01, as instructed.
- For training the network parameters, I wrote the *SGD_Q2* function, which trains the inputted network parameters for the training data and labels using the stochastic gradient descent algorithm with added momentum,

$$\Delta W(n) = -\eta \frac{\partial J}{\partial W} + \alpha \Delta W(n-1) \quad (2)$$

- The embedding layer is a $(250, D)$ shaped matrix and is followed by linear activation.
- The hidden layer is outputs are subjected to sigmoid activation and finally, softmax activation is applied to the results of the output layer.
- Epochs are subdivided into mini-batches of desired sizes, and training is tracked via the value of cost function and the model prediction accuracies.
- At the end of every epoch, the validity of training is checked by tracking the cost value and the accuracies the of model against a validation data set that the network is not trained for.
- The training is stopped based on the cost against the validation data.

This procedure is repeated for three sets of embedding and hidden layer sizes: $(32, 256)$, $(16, 128)$, and $(8, 64)$ respectively. There are some remarks to be made about the training process.

- Training and validation costs for all models start at quite high categorical cross-entropy costs (Around 4 – 4.5 at the end of first epoch) and none of the models achieve particularly good cost values.
- $J_{val} = 2.8418$ and $J_{train} = 2.6819$ at the end of trainings was the best validation cost achieved for the parameters,

$$L_{embed} = 32$$

$$L_{hidden} = 256$$

$$\text{Learning Rate} = 0.15$$

$$\text{Batch Size} = 200$$

$$\alpha = 0.85$$

- Validation accuracy metric never achieved above 23.7% for any of the layer sizes that I experimented with.

3.c) Results for The Testing Data and Discussion

Then, the models accuracy and learning capabilities are studies by looking at the most probable ten predictions of the model for five randomly selected trigrams from the testing data set. Here are the results:

Trigram: ['did', 'nt', 'know'] \rightarrow **True Value:** 'me'

Model Parameters	Top Predictions
$L_{embed} = 32, L_{hidden} = 256$	['.', '??', ',', 'the', 'for', 'with', 'in', 'here', 'at', 'me']
$L_{embed} = 16, L_{hidden} = 128$	['.', '??', ',', 'the', 'for', 'in', 'here', 'now', 'there', 'that']
$L_{embed} = 8, L_{hidden} = 64$	['.', '??', ',', 'the', 'to', 'that', 'for', 'it', 'here', 'in']

Trigram: ['what', 'is', 'best'] \rightarrow **True Value:** 'here'

Model Parameters	Top Predictions
$L_{embed} = 32, L_{hidden} = 256$	['.', ',', '??', 'at', 'in', 'now', 'here', 'the', 'for', 'with']
$L_{embed} = 16, L_{hidden} = 128$	['.', ',', 'here', '??', 'in', 'at', 'there', 'now', 'the', 'going']
$L_{embed} = 8, L_{hidden} = 64$	['.', '??', ',', 'going', 'for', 'out', 'in', 'over', 'here', 'to']

Trigram: ['just', 'a', 'little'] \rightarrow **True Value:** ', '

Model Parameters	Top Predictions
$L_{embed} = 32, L_{hidden} = 256$	['.', ',', 'we', 'you', 'they', '??', 'he', 'the', 'and', 'i']
$L_{embed} = 16, L_{hidden} = 128$	['.', 'he', 'we', 'she', 'i', '??', 'it', ',', 'that', 'you']
$L_{embed} = 8, L_{hidden} = 64$	['.', 'he', 'over', 'about', '??', ',', 'i', 'it', 'and', 'going']

Trigram: ['here', 'with', 'him'] \rightarrow **True Value:** 'for'

Model Parameters	Top Predictions
$L_{embed} = 32, L_{hidden} = 256$	['.', 'said', 's', 'was', 'is', 'does', 'did', 'has', 'says', 'would']
$L_{embed} = 16, L_{hidden} = 128$	['said', '.', 'was', 'says', 'did', 'does', 'he', ',', 'has', 'is']
$L_{embed} = 8, L_{hidden} = 64$	['are', 'can', '.', 'did', 'do', '??', 's', 'is', 'said', 'will']

Trigram: ['so', 'what', 's'] \rightarrow **True Value:** 'that'

Model Parameters	Top Predictions
$L_{embed} = 32, L_{hidden} = 256$	['.', ',', 'he', 'i', 'they', 'you', 'it', 'we', '?', 'she']
$L_{embed} = 16, L_{hidden} = 128$	['.', 'he', ',', 'i', 'said', '?', 'you', 'she', 'it', 'was']
$L_{embed} = 8, L_{hidden} = 64$	['.', ',', '?', 'i', 'what', 'about', 'over', 'there', 'you', 'we']

One common problem observed in all three models' predictions is that in all three cases of model dimensions, the model seems to mostly predict '.' as its primary guess, followed by ',' and '?'. This is caused because of

This behavior is caused by the model over-fitting of the model to the specific labels that are disproportionately abundant in the training set, mainly the labels that correspond to the dictionary elements '.', ',', '?' and a few such characters. A few improvements could be done in order to get better results:

- Using a more balanced training data set would likely help.
- Implementing a pre-trained embedding layer could allow model to learn the correlation between the trigram words better.
- Rearranging the initial data to implement longer windows, like 4-gram or 5-gram instead of the trigram, can make the model predictions more accurate.
- Rather than using only feed-forward layers, implementing a recurrent layer, like LSTM, improve the model predictions by accounting for the sequence of the inputted word tokens.

Question 3.

In this question, three different recurrent neural networks, along with MLP layers, will be implemented to process measurements from three sensors of human activity that are 150 time steps long in order to predict what action the person is doing at that moment. Through the parts of the question, only the type of recurrent layer will be changed. Therefore, I will begin by going through the common processes that take place:

- All weights and biases of the recurrent layer are initialized with Xavier initialization, where the values are sampled from the uniform distribution between $[-w_0, w_0]$ where w_0 is defined as,

$$w_0 = \sqrt{\frac{6}{L_{pre} + L_{post}}} \quad (3)$$

- A multi-layer perceptron of desired amounts of layers and sizes is created through *initSeqWb* function. Weights and biases of MLP are also initiated with Xavier initialization.
- The forward propagation through the recurrent layer is carried out with the corresponding forward propagation function for the type of the recurrent layer: *recurrentForward* for simple recurrent layer, *lstmForward* for LSTM layer, and *gruForward* for GRU layer. This outputs the final activation at the last time step h_{last} and a cache of all other calculation that have been carried out, which is needed for backpropagation.
- The last activation from the recurrent layer is passed into the MLP layer through the function *sequentialForward*.
- All activations for the forward propagation are chosen as *tanh*.
- For the final layer of MLP, *softmax* activation is used.
- The function *sequentialBackward* is called for backpropagating through the MLP. It return the cross-entropy cost of the batch, the derivative of the MLP layer input with respect to the cost dA_0 , and the gradients of the MLP parameters.
- The gradients of the recurrent layer parameters is calculated by the corresponding backpropagation functions: *recBackward* for simple recurrent layer, *lstmBackward* for LSTM layer, and *gruBackward* for GRU layer.
- Recurrent layers make multiple passes over the same weights. Backpropagation through the recurrent layer is carried out by the backpropagation through time algorithm:

$$\Delta w_{ij} = -\eta \frac{\partial J_{total}}{\partial w_{ij}} = -\eta \sum_{t=t_0}^{t_{end}} \frac{\partial J(t)}{\partial w_{ij}} \quad (4)$$

where η is the learning rate and $J(t)$ is the loss

- The *SGD-Q3* function carries out the forward propagation and backpropagation for the specified recurrent layer type and implements stochastic gradient descent algorithm with an added momentum term:

$$\Delta W(n) = -\eta \frac{\partial J_{total}}{\partial W} + \alpha \Delta W(n-1) \quad (5)$$

where α is the momentum term, and determined how much of the previous updates' gradients are conserved. The weights and biases are updated with this calculated change term,

$$W(n) = W(n-1) + \Delta W(N) \quad (6)$$

- *SGD-Q3* function allows for setting desired amount of batch size, learning rate, momentum multiplier α .
- The training is stopped when the specified validation loss is achieved or epoch number reaches maximum epochs specified. However, for displaying the experiment results, this threshold is set especially low to see how the model trains for all 50 epochs.
- On top of the instructions, I added a parameter *max_lookback_distance* which is used to truncate how many time steps the BPTT algorithm considers. By default, the BPTT calculates the gradients for all of the time steps.

To maintain consistency, the following training parameters are used for all three recurrent cell types:

- Recurrent layer hidden size is set to be 128.
- MLP layer sizes are chosen to be [3, 128, 64, 64, 6], including the input and the output layer sizes.
- The other training hyper-parameters are set as,

$$\begin{aligned} \text{Batch Size} &= 32 \\ \text{Learning Rate} &= 0.01 \\ \alpha &= 0.85 \end{aligned}$$

3.a) Simple RNN Implementation

Simple recurrent neural network carries the previous cell activation to the next time step through the following equation:

$$\begin{aligned} h_t &= \sigma_h(W_{ih}x_t + W_{hh}h_{t-1} + b_{ih}) \\ y_t &= \sigma_y(W_{oh}h_t + b_{oh}) \end{aligned}$$

where $W_{ih, hh, oh}$ are the connection weights, x_t is the input at time t , and h_t is the output at time t . For my experiments, I took the initial hidden activation as $h_0 = 0$. An illustration of the connections is shown in figure 7.

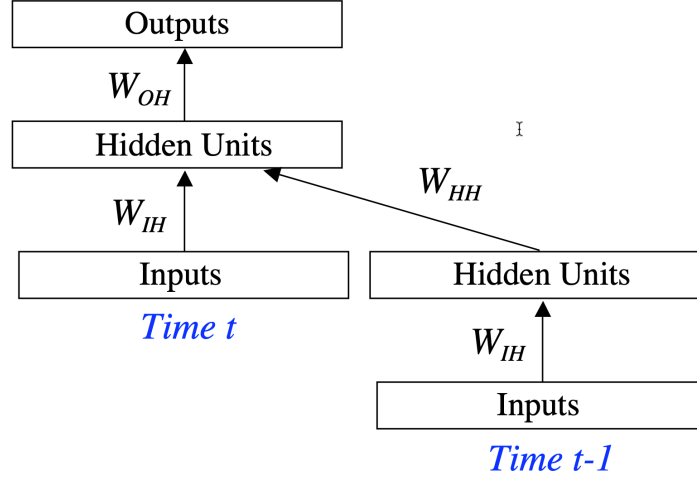
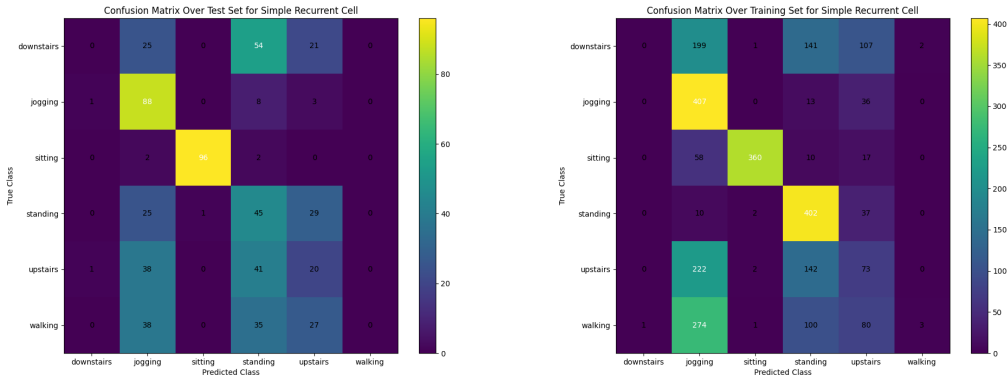


Figure 7: Simple recurrent neural network unfolded in time.

I observed that the training of the simple recurrent network was highly unstable. Model prediction accuracy was poor in all training validation and testing data sets. In the recurrent layer weights, multiple backward passes were calculated as back-propagation through time algorithm intended, and these gradient are summed up. Due to the cumulative affect of these long range gradients, I observed that the final gradient of the recurrent layer weights were either vanishing or, and more dangerously, blowing up. I suspect that this caused the recurrent layer to not learn effectively and since it comes first, among the layers, causes the whole training process to be unstable. Plots of these findings can be seen in figures 8 and 9.



(a) Confusion matrix for testing set.

(b) Confusion matrix for training set.

Figure 9: The confusion matrices of the simple RNN model predictions for the training and testing datasets.

It is worth mentioning that this performance can be enhanced in a relatively simple way. I experimented with truncating the back-propagation through time algorithm to only 20 time steps and doing so improved the model accuracy drastically. In figures 10

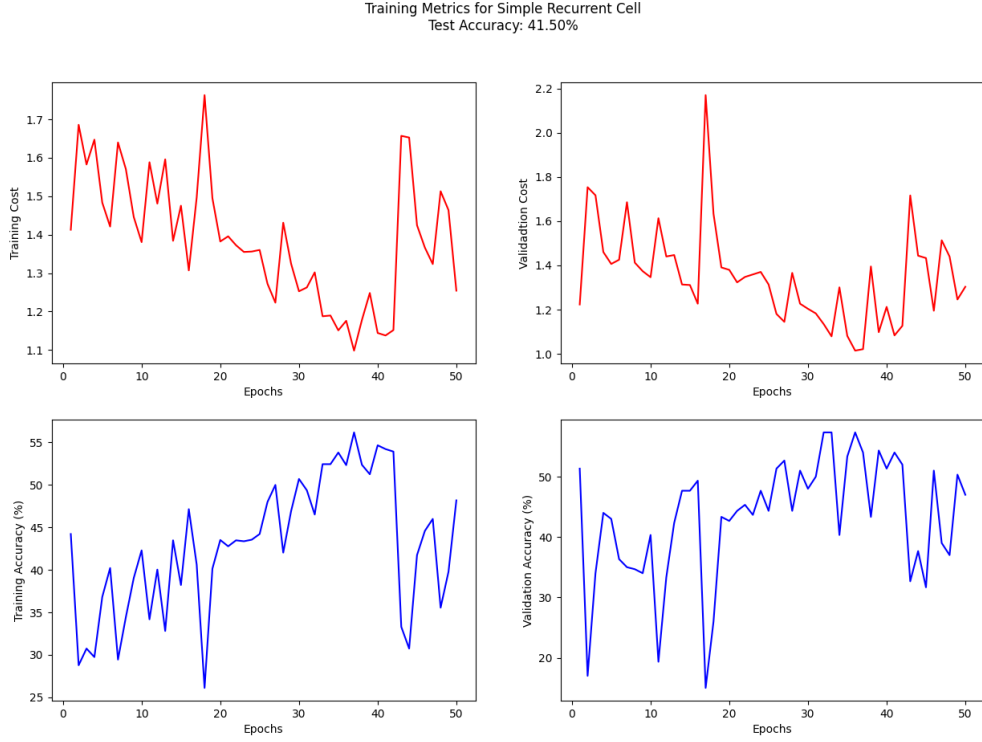
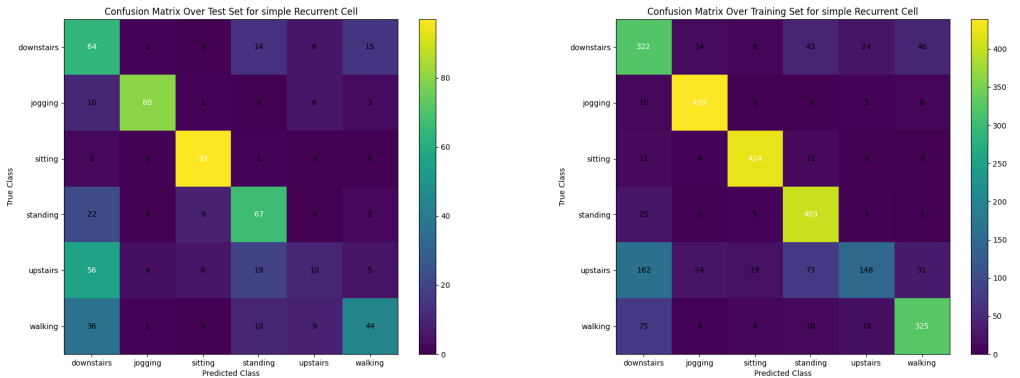


Figure 8: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.

and 11 are the results for simple recurrent cell with BPTT truncation to 20 time steps.



(a) Confusion matrix for testing set.

(b) Confusion matrix for training set.

Figure 11: The confusion matrices of the simple RNN model predictions for the training and testing datasets with BPTT truncation.

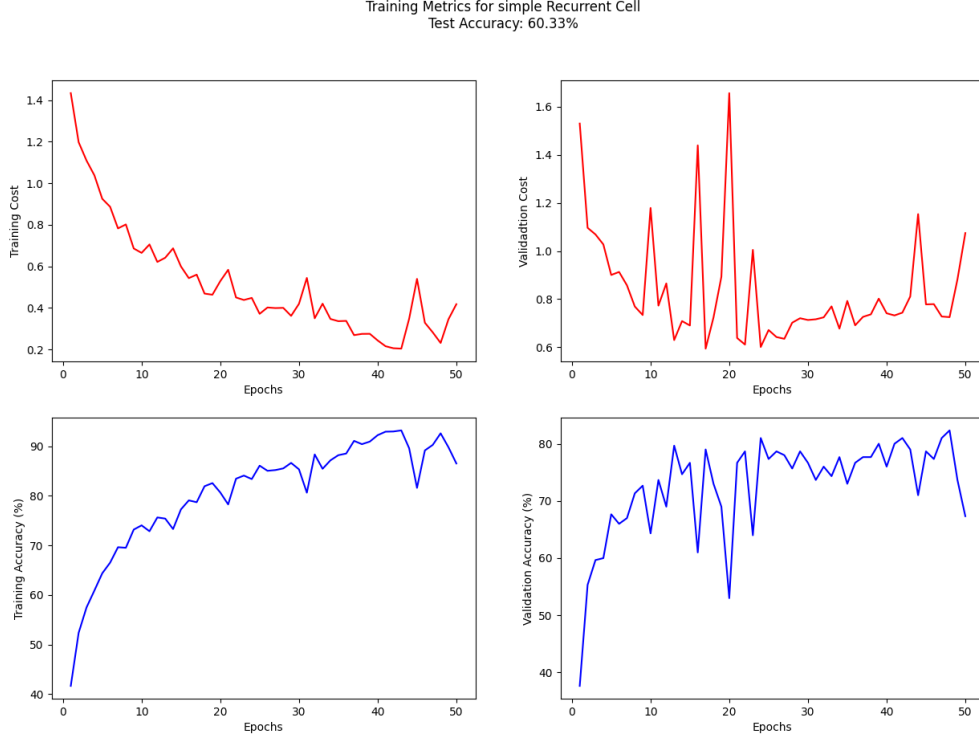


Figure 10: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number with BPTT truncation.

3.b) LSTM Implementation

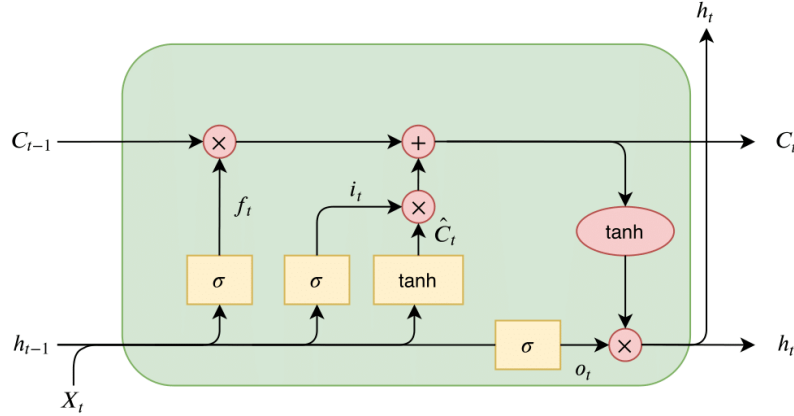


Figure 12: An illustration of the LSTM cell and its inner mechanisms.

Long-Short term memory (LSTM) cell was suggested as an improvement to the recurrent neural network to prevent the exploding/vanishing gradients problem. LSTM cell implements gating in order to regulate how much information is passed through to the different parts of the cell, as seen in figure 12. Here are the equations that take place for propagating a single time step through the LSTM cell:

$$\begin{aligned}
\text{Forget Gate: } f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
\text{Input Gate: } i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
\text{Output Gate: } o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\text{Candidate Cell state: } \tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
\text{Updated Cell State: } c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
\text{Hidden State: } h_t &= o_t \odot \sigma_h(c_t)
\end{aligned} \tag{7}$$

where $W_{f,i,o,c}$ and $U_{f,i,o,c}$ make up the network weights, x_t is the input, c_t is the cell state, h_t is the hidden activation (output) of the cell at time t , and σ_g and σ_c are activations originally taken as *sigmoid* and *tanh* functions respectively.

LSTM cell gave a lot better results than the simple recurrent cell, getting up to 80% accuracy on the test and validation data. Gates that are implemented in the LSTM cell allows it to capture long time dependencies, without becoming unstable. For simple recurrent cell, truncation was necessary to get proper results. However with LSTM cell, the forget gate helps regulate the amount of information flow, making such a practice unnecessary. In figures 13 and 14 are the training and testing results of the model with LSTM cell. Although the LSTM model performs well, after achieving around 85% accuracy on the validation set, the model diverges and the accuracy drops to around 50% accuracy on the validation set in a few epochs. To prevent this, I stopped the training early.

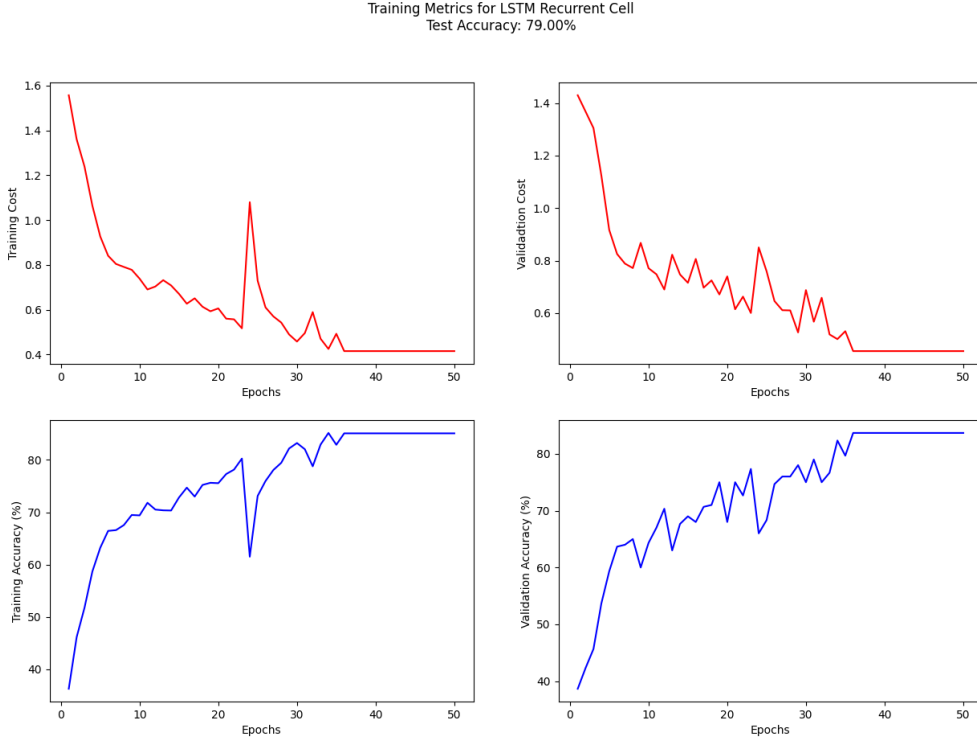
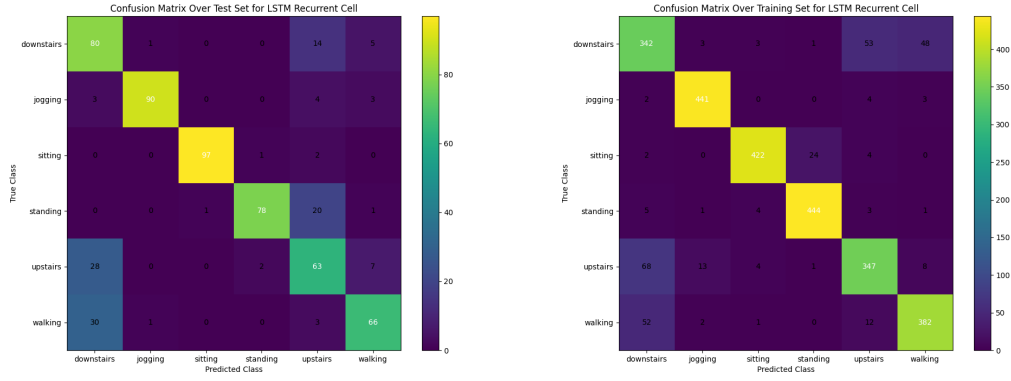


Figure 13: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.



(a) Confusion matrix for testing set.

(b) Confusion matrix for training set.

Figure 14: The confusion matrices of the model with LSTM cell's predictions for the training and testing datasets.

3.c) GRU Implementation

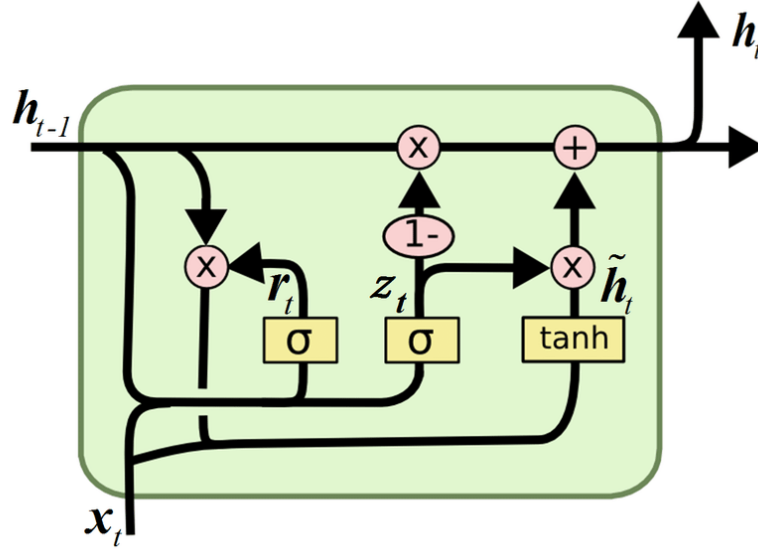


Figure 15: An illustration of the GRU cell and its inner mechanisms.

Gated recurrent unit, or GRU for short, was introduced as a simplification of LSTM cell with fewer parameters. It has a similar structure with LSTM, having derivatives of input and forget gates, as seen in figure 15. However, it lacks the output gate. The following equations describe the information flow in a GRU cell.

$$\begin{aligned}
 \text{Update gate: } z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
 \text{Reset gate: } r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
 \text{Candidate Output: } \hat{h}_t &= \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
 \text{Output: } h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t
 \end{aligned} \tag{8}$$

where $W_{z,r,h}$ and $U_{z,r,h}$ are the recurrent cell weights, h_t is the output, and x_t is the input at time t .

GRU cell is observed to perform almost as good as the LSTM model. But LSTM has outperformed the GRU model for my experiments. Regardless, GRU has had some advantages over the LSTM model:

- Due to GRU having less parameters, it trained faster.
- Also having to do with GRU having less parameters, the cache values that are created in the forward propagation hold less volume in the memory.
- Backpropagation is easier to implement and faster.

In figures 16 and 17, the training and test performance of GRU cell can be seen.

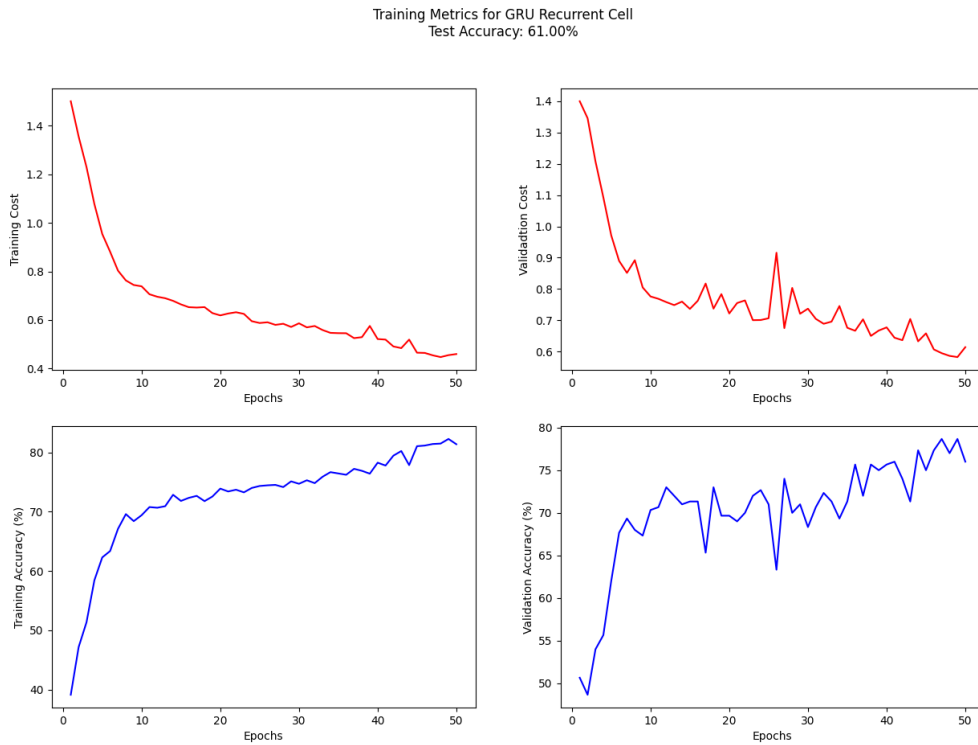
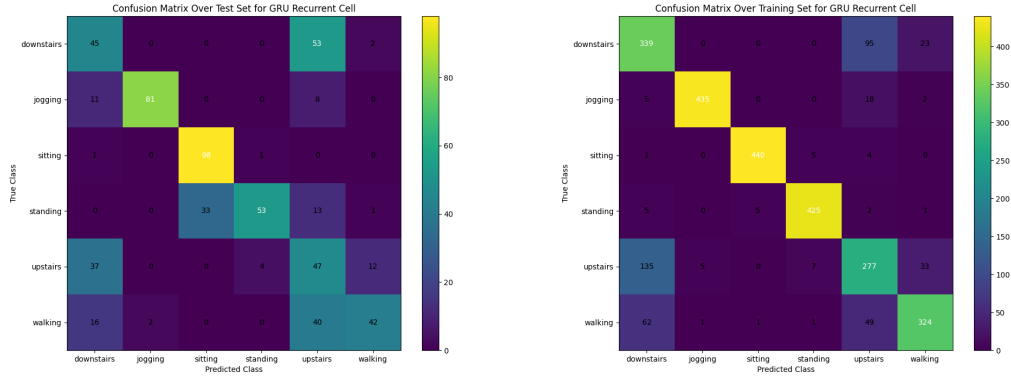


Figure 16: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.



(a) Confusion matrix for testing set. (b) Confusion matrix for the training set.

Figure 17: The confusion matrices of the model with GRU cell's predictions for the training and testing datasets.

As a small side note about question-3, all three of these recurrent networks can be trained to above 80% accuracy on the testing data set by following a more precise and detailed training schedule:

- For the simple recurrent network, it is mandatory to use BPTT truncating for the model to converge.
- First, dividing the training over multiple checkpoints both helps to maintain current performance at desired intervals, 10 epochs works fine.
- High learning rate and high α values are good for training the model from scratch. However, they can cause taking too big steps when the model is near the minima. Therefore, using a smaller learning rate and smaller α value works for fine-tuning the model.
- After experimenting with an initial training with learning rate = 0.01 and $\alpha = 0.85$ for course training and another, shorter training with learning rate = 0.001 and $\alpha = 0.35$, I was able to get to at least 80% accuracy on the testing data for all three types of recurrent networks.

Appendix: Code

Listing 1: Python Code from File

```
1 # Necessary imports
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import h5py
5
6
7 #####
8 ##### CODE FOR Q-1 #####
9 #####
10
11 ## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION
12
13 def sigmoid(X):
14     """Returns the sigmoid function of the array X element-wise.
15
16         sigmoid(X) = 1 / (1 + exp(-X))
17
18     Args:
19         X (np.array.ndarray): Input array.
20
21     Returns:
22         float: Sigmoid of X
23     """
24     return 1 / (1 + np.exp(-X))
25
26 def sigmoid_backward(X):
27     """The derivative of the sigmoid function for back-
28         propagation.
29
30         sigmoid_backward(X) = exp(-X) / ((1 + exp(-X))^2)
31
32     Args:
33         X (np.array.ndarray): Input array
34
35     Returns:
36         _type_: The derivative of sigmoid at X
37     """
38     return np.exp(-X) / ((1 + np.exp(-X))**2)
39
40 def KL_div_ber(P, Q):
41     """Returns the KL divergence between Bernoulli random
42         variables with means P and Q.
43
44         KL-div = (P * log(P / Q)) + ((1-P) * log((1-P) /
45             (1-Q)))
46
47     Args:
48         P (float): Must be between 0 and 1.
```

```

46         Q (float): Must be between 0 and 1.
47
48     Returns:
49         float: The KL divergence value.
50     """
51     return (P * np.log(P / Q)) + ((1-P) * np.log((1-P) / (1-Q)))
52
53 def grad_KL_div_ber(P, Q):
54     """The derivative of the KL divergence of Bernoulli random
55         variables with means P and Q, with respect to Q.
56
57         grad_KL_div_ber = (-P / Q) + ((1-P) / (1-
58             Q))
59
60     Args:
61         P (float): Must be between 0 and 1.
62         Q (float): Must be between 0 and 1.
63
64     Returns:
65         float: The derivative of KL divergence with respect to Q.
66     """
67     return (-P / Q) + ((1-P) / (1-Q))
68
69 def img_to_flat(data):
70     """Takes an set of images, with sample dimention at last
71         dimention, and flattens the images.
72
73     Args:
74         data (np.array.ndarray): Images with shape: (width,
75             height, samples)
76
77     Returns:
78         np.array.ndarray: Flattened images with shape: (width *
79             height, samples)
80     """
81     sample_size = data.shape[2]
82     flat_size = data.shape[0] * data.shape[1]
83     return np.reshape(data, (flat_size, sample_size))
84
85 def flat_to_img(data, img_shape):
86     """Takes a set of data samples and reshapes the first
87         dimention to the desired image shape
88
89     Args:
90         data (np.array.ndarray): Data with shape: (data, samples)
91         img_shape (array): Desired image shape: (width, height)
92
93     Raises:
94         ValueError: It must hold that -> heights * width = data
95
96     Returns:

```

```

91         np.array.ndarray: Data reshaped to images of shape: (
           width, height, samples)
92     """
93     if data.shape[0] != img_shape[0] * img_shape[1]:
94         raise ValueError('Need the shapes to match.')
95
96     sample_size = data.shape[1]
97     return np.reshape(data, (img_shape[0], img_shape[1],
           sample_size))
98
99 # Utility functions for creating the autoencoder network.
100 def init_AE_Wb(input_size, hidden_size):
101     """Initializes the weights and biases for a single hidden
           layer autoencoder network. Uses Xavier initialization
           technique.
102
103     Args:
104         input_size (int): The input size, same as the output size
105         hidden_size (int): The hidden layer size
106
107     Returns:
108         array: array that contains the weights and biases: [W1,
           W2, b1, b2]
109     """
110     # Defining w0
111     w0 = np.sqrt(6 / (input_size + hidden_size))
112
113     # Random initialization of the parameters
114     W1 = np.random.uniform(-w0, w0, (hidden_size, input_size))
115     W2 = np.random.uniform(-w0, w0, (input_size, hidden_size))
116     b1 = np.random.uniform(-w0, w0, (hidden_size, 1))
117     b2 = np.random.uniform(-w0, w0, (input_size, 1))
118
119     Wb = [W1, W2, b1, b2]
120
121     return Wb
122
123 def aeCost(W_e, data, params):
124     """Calculates the error of the autoencoder network with
           parameters W_e, for the given data.
125
126     Args:
127         W_e (array): Array containing the network parameters: [W1
           , W2, b1, b2]
128         data (np.array.ndarray): The input data. Must be in shape
           : (L_in, samples)
129         params (array): Array containing the additional
           information: [L_in, L_hid, cost_lambda, cost_beta,
           cost_rho]
130
131     Returns:

```

```

132     [float, dict]: array containing the net cost and a
133         dictionary containing the gradients of
134         cost with respect to network parameters and activations:
135         [J, J_grad]
136     """
137
138     # Unpacking the network parameters
139     W1, W2, b1, b2 = W_e
140
141     batch_size = data.shape[1]
142
143     L_in = params[0]
144     L_hid = params[1]
145     cost_lambda = params[2]
146     cost_beta = params[3]
147     cost_rho = params[4]
148
149     # Forward propagation calculations
150     A0 = data
151     Z1 = np.matmul(W1, A0) + b1
152     A1 = sigmoid(Z1)
153     Z2 = np.matmul(W2, A1) + b2
154     A2 = sigmoid(Z2)
155
156     # Calculating the loss
157     J_tto = (1 / (2 * batch_size)) * np.sum( (A2 - A0)**2 )
158     J_Tykhonov = (cost_lambda / 2) * (np.sum( (W1)**2 ) + np.sum(
159         (W2)**2 ))
160     J_KL = cost_beta * np.sum(KL_div_ber(cost_rho, A1)) /
161         batch_size
162     J = J_tto + J_Tykhonov + J_KL
163
164     # Calculating the derivatives
165     dA2 = (A2 - A0) #/ batch_size
166     dZ2 = dA2 * sigmoid_backward(Z2)
167     dW2 = np.matmul(dZ2, A1.T) / batch_size + (cost_lambda * W2)
168     db2 = np.sum(dZ2, axis=1, keepdims=True) / batch_size
169     dA1 = np.matmul(W2.T, dZ2) + (cost_beta * grad_KL_div_ber(
170         cost_rho, A1) / batch_size)
171     dZ1 = dA1 * sigmoid_backward(Z1)
172     dW1 = np.matmul(dZ1, A0.T) / batch_size + (cost_lambda * W1)
173     db1 = np.sum(dZ1, axis=1, keepdims=True) / batch_size
174
175     J_grad = {
176         'dA2': dA2,
177         'dZ2': dZ2,
178         'dW2': dW2,
179         'db2': db2,
180         'dA1': dA1,
181         'dZ1': dZ1,
182         'dW1': dW1,

```

```

178         'db1':db1,
179     }
180     return [J, J_grad]
181
182 def solver(data, W_e, params, lr, batch_size, epochs, beta1=0.9,
183           beta2=0.999, epsilon=1e-8, verbose=True):
184     """Trains the network with parameters W_e with the given data
185         using the Adam algorithm.
186
187     Args:
188         data (np.array.ndarray): Input data to be used for
189             training. Shape must be (L_in, samples)
190         W_e (array): Array with the initial network parameters: [
191             W1, W2, b1, b2]
192         params (array): Array containing the additional
193             information: [L_in, L_hid, cost_lambda, cost_beta,
194             cost_rho]
195         lr (float): Learning rate for training. Advised to be
196             below 1e-3
197         batch_size (int): Number of samples used per parameter
198             updating
199         epochs (int): Number of iterations over the data
200         beta1 (float, optional): Beta1 value for Adam optimizer.
201             Defaults to 0.9.
202         beta2 (float, optional): Beta2 value for Adam optimizer.
203             Defaults to 0.999.
204         epsilon (float, optional): Epsilon value for Adam
205             optimizer. Defaults to 1e-8.
206         verbose (bool, optional): For verbosing the training
207             progress. Defaults to True.
208
209     Returns:
210         [[W1, W2, b1, b2], history]: The trained network
211             parameters and the history of loss value for progress
212             tracking
213
214     """
215     # Unpackinng the parameters
216     W1, W2, b1, b2 = W_e
217
218     # Initializing the momentum and velocity for Adam
219     m_W1, m_W2, m_b1, m_b2 = 0, 0, 0, 0
220     v_W1, v_W2, v_b1, v_b2 = 0, 0, 0, 0
221
222     # Iteration number for Adam
223     t = 0
224
225     sample_number = data.shape[1]
226     passes_per_epoch = sample_number // batch_size
227
228     # Initializing the loss history
229     loss_history = np.zeros((epochs, 1))

```

```

215
216 print('Starting training...\n')
217
218 for i in range(epochs):
219
220     epoch_loss = 0
221
222     if verbose:
223         print(f'Epoch: [{i+1}/{epochs}] -> ', end='')
224
225     for j in range(passes_per_epoch):
226         t += 1
227
228         # Calculating the cost and the gradients for the
229         # batch
230         J, J_grad = aeCost(W_e, data[:, j*batch_size:(j+1)*
231             batch_size], params)
232         epoch_loss += J
233
234         # Updating the moemntum values
235         m_W1 = beta1 * m_W1 + (1 - beta1) * J_grad['dW1']
236         m_W2 = beta1 * m_W2 + (1 - beta1) * J_grad['dW2']
237         m_b1 = beta1 * m_b1 + (1 - beta1) * J_grad['db1']
238         m_b2 = beta1 * m_b2 + (1 - beta1) * J_grad['db2']
239
240         # Updating the velocity values
241         v_W1 = beta2 * v_W1 + (1 - beta2) * (J_grad['dW1'] **
242             2)
243         v_W2 = beta2 * v_W2 + (1 - beta2) * (J_grad['dW2'] **
244             2)
245         v_b1 = beta2 * v_b1 + (1 - beta2) * (J_grad['db1'] **
246             2)
247         v_b2 = beta2 * v_b2 + (1 - beta2) * (J_grad['db2'] **
248             2)
249
250         # Normalizing the momentum
251         m_W1_hat = m_W1 / (1 - beta1 ** t)
252         m_W2_hat = m_W2 / (1 - beta1 ** t)
253         m_b1_hat = m_b1 / (1 - beta1 ** t)
254         m_b2_hat = m_b2 / (1 - beta1 ** t)
255
256         # Normalizing the velocity
257         v_W1_hat = v_W1 / (1 - beta2 ** t)
258         v_W2_hat = v_W2 / (1 - beta2 ** t)
259         v_b1_hat = v_b1 / (1 - beta2 ** t)
260         v_b2_hat = v_b2 / (1 - beta2 ** t)
261
262         # Updating the weights and biases
263         W1 -= lr * m_W1_hat / (np.sqrt(v_W1_hat) + epsilon)
264         W2 -= lr * m_W2_hat / (np.sqrt(v_W2_hat) + epsilon)
265         b1 -= lr * m_b1_hat / (np.sqrt(v_b1_hat) + epsilon)

```



```

260         b2 -= lr * m_b2_hat / (np.sqrt(v_b2_hat) + epsilon)
261
262         # Updating W_e for the next batch of training
263         W_e = [W1, W2, b1, b2]
264
265     # Same training loop for the left-over samples that are
266     # left to the last batch
267     if sample_number % batch_size != 0:
268         t += 1
269
270         J, J_grad = aeCost(W_e, data[:, (passes_per_epoch*
271             batch_size):], params)
272         epoch_loss += J
273
274         m_W1 = beta1 * m_W1 + (1 - beta1) * J_grad['dW1']
275         m_W2 = beta1 * m_W2 + (1 - beta1) * J_grad['dW2']
276         m_b1 = beta1 * m_b1 + (1 - beta1) * J_grad['db1']
277         m_b2 = beta1 * m_b2 + (1 - beta1) * J_grad['db2']
278
279         v_W1 = beta2 * v_W1 + (1 - beta2) * (J_grad['dW1'] **
280             2)
281         v_W2 = beta2 * v_W2 + (1 - beta2) * (J_grad['dW2'] **
282             2)
283         v_b1 = beta2 * v_b1 + (1 - beta2) * (J_grad['db1'] **
284             2)
285         v_b2 = beta2 * v_b2 + (1 - beta2) * (J_grad['db2'] **
286             2)
287
288         m_W1_hat = m_W1 / (1 - beta1 ** t)
289         m_W2_hat = m_W2 / (1 - beta1 ** t)
290         m_b1_hat = m_b1 / (1 - beta1 ** t)
291         m_b2_hat = m_b2 / (1 - beta1 ** t)
292
293         v_W1_hat = v_W1 / (1 - beta2 ** t)
294         v_W2_hat = v_W2 / (1 - beta2 ** t)
295         v_b1_hat = v_b1 / (1 - beta2 ** t)
296         v_b2_hat = v_b2 / (1 - beta2 ** t)
297
298         W1 -= lr * m_W1_hat / (np.sqrt(v_W1_hat) + epsilon)
299         W2 -= lr * m_W2_hat / (np.sqrt(v_W2_hat) + epsilon)
300         b1 -= lr * m_b1_hat / (np.sqrt(v_b1_hat) + epsilon)
301         b2 -= lr * m_b2_hat / (np.sqrt(v_b2_hat) + epsilon)
302
303         W_e = [W1, W2, b1, b2]
304
305     epoch_loss = (epoch_loss * batch_size) / sample_number
306     loss_history[i] = epoch_loss
307
308     if verbose:
309         print(f'Epoch Loss: {epoch_loss:.4f}')

```

```

305     print('\nTraining completed.')
306
307     return [[W1, W2, b1, b2], loss_history]
308
309 def predict(W_e, data):
310     """Make predictions for the data with the network with
311         parameters W_e
312
313     Args:
314         W_e (array): Array with the initial network parameters: [
315             W1, W2, b1, b2]
316         data (np.array.ndarray): Data in shape: (L_in, samples)
317
318     Returns:
319         np.array.ndarray: The network predictions of shape: (
320             L_out, samples)
321     """
322     W1, W2, b1, b2 = W_e
323
324     Z1 = np.matmul(W1, data) + b1
325     A1 = sigmoid(Z1)
326
327     Z2 = np.matmul(W2, A1) + b2
328     A2 = sigmoid(Z2)
329
330     return A2
331
332 # The function to perform the question-1 code
333 def q1():
334     """ PART - 1.A
335     #####
336     # extracting the data from the data1.h5 file
337     print('\nBegining Question-1 Part-A...\n')
338
339     with h5py.File('data1.h5', 'r') as file:
340         data_key = list(file.keys())[0]
341         data = np.array(file[data_key])
342
343     print('\nInitial Data Shape:', np.shape(data))
344
345     # Carrying the channels to last
346     data_channels_last = np.transpose(data, (0, 2, 3, 1))
347     data_channels_last_scaled = (data_channels_last -
348         data_channels_last.min()) / ((data_channels_last -
349         data_channels_last.min()).max())
350
351     print('Data With Channels-Last:', data_channels_last_scaled.
352         shape)
353     print('Scaled Data With Channels Max:',
354         data_channels_last_scaled.max())

```

```

348     print('Scaled Data With Channels Min:',
349           data_channels_last_scaled.min())
350
351     # scaling the data for the lumousity level
352     data_gs = np.average(data, axis=1, weights=(0.2126, 0.7152,
353           0.0722))
354
355     print('\nData Shape After Lumousity Scaling:', data_gs.shape)
356     print('Gray-Scale Data Maximum Value:', data_gs.max())
357     print('Gray-Scale Minumum Value:', data_gs.min())
358
359     # Normalize the data by clipping the beyond three standart
360     # deviations
361     data_gs_mean = np.mean(data_gs)
362     data_gs_std = np.std(data_gs)
363
364     print('\nData Average Value:', data_gs_mean)
365     print('Data Standard Deviation:', data_gs_std)
366
367     data_clip_limit_below = data_gs_mean - 3 * data_gs_std
368     data_clip_limit_above = data_gs_mean + 3 * data_gs_std
369
370     data_gs_normalized = np.clip(data_gs, data_clip_limit_below,
371     data_clip_limit_above)
372
373     print('\nClipped Data Shape:', data_gs_normalized.shape)
374     print('Clipped Data Maximum Value:', data_gs_normalized.max()
375     )
376     print('Clipped Data Minumum Value:', data_gs_normalized.min()
377     )
378
379     # Scale the data to [0.1, 0.9] interval
380     data_gs_norm_max = data_gs_normalized.max()
381     data_gs_norm_min = data_gs_normalized.min()
382
383     data_gs_scaled = (data_gs_normalized / ((data_gs_norm_max -
384     data_gs_norm_min) / 0.8)) \
385     - ((data_gs_normalized / ((data_gs_norm_max - data_gs_norm_min
386     ) / 0.8)).min() - 0.1)
387
388     print('\nScaled Data Shape:', data_gs_scaled.shape)
389     print('Scaled Maximum Value:', data_gs_scaled.max())
390     print('Scaled Minumum Value:', data_gs_scaled.min())
391
392     # Displaying the data
393     sample_count = data_gs_scaled.shape[0]
394
395     # Selecting the random samples

```

```

391 random_samples = np.random.randint(sample_count, size=200)
392
393 rows = 10
394 columns = 20
395
396 # Displaying the colored samples
397 fig_gs = plt.figure(1, figsize=(16, 8))
398 plt.title('Selected Gray-Scale Data Samples')
399
400 for i in range(1, rows * columns + 1):
401     fig_gs.add_subplot(rows, columns, i)
402     plt.imshow(data_gs_scaled[random_samples[i-1]], cmap='
        gray')
403     plt.axis('off')
404
405 plt.subplots_adjust(hspace=0.01, wspace=0.01)
406 plt.gca().set_axis_off()
407 plt.savefig('Q1_A_gray_scale_images.png')
408
409 # Displaying the gray-scale, normalized samples
410 fig_colored = plt.figure(2, figsize=(16, 8))
411 plt.title('Selected Colored Data Samples')
412
413 for i in range(1, rows * columns + 1):
414     fig_colored.add_subplot(rows, columns, i)
415     plt.imshow(data_channels_last_scaled[random_samples[i
        -1]])
416     plt.axis('off')
417
418 plt.subplots_adjust(hspace=0.01, wspace=0.01)
419 plt.gca().set_axis_off()
420 plt.savefig('Q1_A_colored_images.png')
421
422 print('--'*50)
423
424 ## PART - 1.B
425 #####
426 # Shuffling the data samples
427 print('\nBegining Question-1 Part-B...\n')
428
429 np.random.shuffle(data_gs_scaled)
430 print(f'\nData shape after shuffle: {data_gs_scaled.shape}')
431
432 # Carrying the samples dimention to last for network
    dimentions
433 data_gs_scaled = np.transpose(data_gs_scaled, (1, 2, 0))
434 print(f'Data shape after reshaping: {data_gs_scaled.shape}')
435
436 # Flattening the images for proper shape.
437 model_input_train = img_to_flat(data_gs_scaled)
438

```

```

439 # Network sizes
440 input_size = 256
441 hidden_size = 64
442
443 # Initializing the weights and biases
444 Wb = init_AE_Wb(input_size, hidden_size)
445
446 # Assigning the cost parameters
447 # Best values for cost_beta and cost_rho are 0.1 and 0.5
    respectively.
448 cost_lambda = 5e-4
449 cost_beta = 0.1
450 cost_rho = 0.5
451 params = [input_size, hidden_size, cost_lambda, cost_beta,
    cost_rho]
452
453 # Batch size and epoch number
454 batch_size = 32
455 epochs = 50
456
457 # Training the network for the training data
458 Wb_trained, history = solver(model_input_train, Wb, params, 1
    e-3, batch_size, epochs)
459
460 # x-axis for the loss history
461 history_x_axis = np.arange(1, history.size+1)
462
463 # Plotting the training losses
464 fig_error = plt.figure(3, figsize=(5,5))
465 plt.title('Training Error Values')
466 plt.xlabel('Epochs')
467 plt.ylabel('Error')
468 plt.plot(history_x_axis, history)
469 plt.savefig('Q1_B_loss_history.png')
470
471
472 ### PART - 1.C
473 #####
474 # Plotting the first layer connection weights
475 print('\nBeginning Question-1 Part-C...\n')
476
477 W1 = Wb_trained[0].transpose()
478
479 # Converting the connection weights from 1-D to 2-D array
480 model_W1_img = flat_to_img(W1, (16, 16))
481 model_W1_img = np.transpose(model_W1_img, (2, 0, 1))
482
483 figure_latent = plt.figure(figsize=(12,12))
484 plt.title(f'lambda = {cost_lambda}, L_hid = {hidden_size}')
485 rows_latent = 8
486 columns_latent = 8

```

```

487
488 # Plotting for every neuron in the hidden layer
489 for i in range(1, rows_latent * columns_latent + 1):
490     figure_latent.add_subplot(rows_latent, columns_latent, i)
491     plt.imshow(model_W1_img[i-1], cmap='gray')
492     plt.axis('off')
493
494 plt.subplots_adjust(hspace=0.1, wspace=0.1)
495 plt.gca().set_axis_off()
496 plt.savefig('Q1_C_connection_weights.png')
497
498
499 ### PART - 1.D
500 #####
501 print('\nBegining Question-1 Part-D...\n')
502
503 L_hid_values = [16, 49, 81]
504 lambda_values = [1e-5, 1e-4, 1e-3]
505
506
507 params = [input_size, hidden_size, cost_lambda, cost_beta,
508           cost_rho]
509
510 for lambda_value in lambda_values:
511     for L_hid_value in L_hid_values:
512         print(f'\nTraining for lambda={lambda_value} | L_hid
513              = {L_hid_value}\n')
514
515         params = [input_size, L_hid_value, lambda_value,
516                  cost_beta, cost_rho]
517
518         Wb = init_AE_Wb(input_size, L_hid_value)
519         Wb_trained, history = solver(model_input_train, Wb,
520                                     params, 1e-3, batch_size, epochs)
521
522         W1 = Wb_trained[0].transpose()
523
524         model_W1_img = flat_to_img(W1, (16, 16))
525         model_W1_img = np.transpose(model_W1_img, (2, 0, 1))
526
527         figure_latent = plt.figure(figsize=(12,12))
528         plt.title(f'lambda={lambda_value} | L_hid = {
529                  L_hid_value}')
530         rows_latent = int(np.sqrt(L_hid_value))
531         columns_latent = int(np.sqrt(L_hid_value))
532
533         for i in range(1, rows_latent * columns_latent + 1):
534             figure_latent.add_subplot(rows_latent,
535                                     columns_latent, i)
536             plt.imshow(model_W1_img[i-1], cmap='gray')
537             plt.axis('off')

```

```

532         plt.subplots_adjust(hspace=0.1, wspace=0.1)
533         plt.gca().set_axis_off()
534         plt.savefig(f'Q1_D_{lambda_value}_{L_hid_value}.png')
535
536
537
538     plt.show()
539
540
541
542 #####
543 ##### CODE FOR Q-2 #####
544 #####
545
546 ## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION
547
548 def sigmoid(X):
549     return 1 / (1 + np.exp(-X))
550
551 def sigmoid_backward(X):
552     return np.exp(-X) / ((1 + np.exp(-X))**2)
553
554 def relu(X, alpha=0.01):
555     return np.where(X>0, X, alpha * X)
556
557 def relu_backward(X, alpha=0.01):
558     return np.where(X>0, 1, alpha)
559
560 def softmax(X, axis=0):
561     max_vals = np.max(X, axis=axis, keepdims=True)
562     e_X = np.exp(X - max_vals)
563     Y = e_X / np.sum(e_X, axis=axis, keepdims=True)
564     return Y
565
566
567 def init_NLP_Wb(dict_size, embed_size, hidden_size, std=0.01):
568     """ Initializes the network parameters for the given
569         embedding layer and hidden layer size.
570
571     Args:
572         dict_size (int): Number of words in the dictionary
573         embed_size (int): Output dimnetion of the embedding layer
574         hidden_size (int): Number of neurons in the hidden layer.
575         std (float, optional): The standard deviation of of the
576             Gaussian distribution from
577             which the parameters are randomly sampled. Defaults to
578             0.01.
579
580     Returns:
581         dict: The dictionary that contains the initialized
582             network parameters.

```

```

579     """
580
581     W_embed = np.random.normal(0, std, (dict_size, embed_size))
582
583     W1 = np.random.normal(0, std, (embed_size, hidden_size))
584     W2 = np.random.normal(0, std, (hidden_size, dict_size))
585
586     b1 = np.random.normal(0, std, (1, hidden_size))
587     b2 = np.random.normal(0, std, (1, dict_size))
588
589     params = {
590         'W_embed': W_embed,
591         'W1': W1,
592         'W2': W2,
593         'b1': b1,
594         'b2': b2
595     }
596
597     return params
598
599 def NLP_forward_pass(params, data, dict_size):
600     """ Performs forward propagation through the network.
601
602     Args:
603         params (dict): The dictionary of the network parameters.
604         data (numpy.array.ndarray): The data to pass through the
605             network.
606         dict_size (int): Number of words in the dictionary
607
608     Returns:
609         dict: A cache of the results from the intermediate steps.
610     """
611     W_embed = params['W_embed']
612     W1 = params['W1']
613     W2 = params['W2']
614     b1 = params['b1']
615     b2 = params['b2']
616
617     batch_size, context_size = data.shape
618     data = np.eye(dict_size)[data]
619
620     A_pre = np.sum(data, axis=1) / context_size
621
622     A0 = np.matmul(A_pre, W_embed)
623
624     Z1 = np.matmul(A0, W1) + b1
625     A1 = sigmoid(Z1)
626
627     Z2 = np.matmul(A1, W2) + b2
628     A2 = softmax(Z2, axis=1)

```



```

629     cache = {
630         'A_pre': A_pre,
631         'A0': A0,
632         'Z1': Z1,
633         'A1': A2,
634         'Z2': Z2,
635         'A2': A2
636     }
637
638     return cache
639
640
641 def nlpCost(params, data, labels, dict_size):
642     """ Calculates the cost with respect to the data and the
643         labels,
644         and the gradient with respect to them.
645
646     Args:
647         params (dict): The dictionary of the network parameters.
648         data (numpy.array.ndarray): The data to calculate cost
649         against.
650         labels (numpy.array.ndarray): The true labels for the
651         data.
652         dict_size (int): Number of words in the dictionary
653
654     Returns:
655         list: A list [J, grads] that contain average cost and
656         gradients with respect to the parameters.
657     """
658
659     W_embed = params['W_embed']
660     W1 = params['W1']
661     W2 = params['W2']
662     b1 = params['b1']
663     b2 = params['b2']
664
665     # converting the data and the labels to one-hot encodings.
666     batch_size, context_size = data.shape
667     data = np.eye(dict_size)[data]
668     labels = np.eye(dict_size)[labels]
669
670     # Forward propagation through the network
671     A_pre = np.sum(data, axis=1)
672
673     A0 = np.matmul(A_pre, W_embed)
674
675     Z1 = np.matmul(A0, W1) + b1
676     A1 = sigmoid(Z1)
677
678     Z2 = np.matmul(A1, W2) + b2
679     A2 = softmax(Z2, axis=1)

```

```

676
677 # Cross-entropy cost calculation
678 J = np.sum(-labels * np.log(A2)) / batch_size
679
680 # Backpropagation
681 dZ2 = (A2 - labels)
682 dW2 = np.matmul(A1.T, dZ2) / batch_size
683 db2 = np.sum(dZ2, axis=0, keepdims=True) / batch_size
684
685 dA1 = np.matmul(dZ2, W2.T)
686 dZ1 = sigmoid_backward(Z1) * dA1
687 dW1 = np.matmul(A0.T, dZ1) / batch_size
688 db1 = np.sum(dZ1, axis=0, keepdims=True) / batch_size
689
690 dA0 = np.matmul(dZ1, W1.T) # = dZ0
691 dW_embed = np.matmul(A_pre.T, dA0) / batch_size
692
693 grads = {
694     'dZ2': dZ2,
695     'dW2': dW2,
696     'db2': db2,
697     'dA1': dA1,
698     'dZ1': dZ1,
699     'dW1': dW1,
700     'db1': db1,
701     'dA0': dA0,
702     'dW_embed': dW_embed
703 }
704 return [J, grads]
705
706
707 def SGD_Q2(params, dict_size, data, labels, val_data, val_labels,
708            lr, batch_size, stop_loss=0.1, max_epochs=50, alpha=0,
709            verbose=True):
710     """
711     Implements stochastic gradient descent (SGD) for training a
712     neural network to predict
713     the fourth word in a sequence based on preceding trigrams.
714     Here are some notes:
715     - SGD_Q2 function implements weight updates using momentum.
716     - Stops training if validation loss falls below 'stop_loss'
717       or 'max_epochs' is reached.
718     - Tracks and reports training and validation accuracy, as
719       well as losses.
720     - Assumes the use of auxiliary functions 'nlpCost' and '
721       NLP_forward_pass' for
722       calculating cost, gradients, and predictions.
723
724     Parameters
725     -----
726     params : dict

```

```

720         Dictionary containing model parameters (weights and
721             biases).
722     dict_size : int
723         Size of the vocabulary.
724     data : numpy.ndarray
725         Training input data, where each row represents a trigram.
726     labels : numpy.ndarray
727         Training labels corresponding to the fourth word in each
728             trigram.
729     val_data : numpy.ndarray
730         Validation input data for monitoring loss and accuracy.
731     val_labels : numpy.ndarray
732         Validation labels corresponding to val_data.
733     lr : float
734         Learning rate for the SGD algorithm.
735     batch_size : int
736         Size of mini-batches for SGD.
737     stop_loss : float, optional
738         Target validation loss for stopping criteria (default is
739             0.1).
740     max_epochs : int, optional
741         Maximum number of training epochs (default is 50).
742     alpha : float, optional
743         Momentum factor to stabilize updates (default is 0).
744     verbose : bool, optional
745         If True, prints progress and metrics at each epoch (
746             default is True).
747
748     Returns
749     -----
750     list
751         A list containing:
752         - Updated model parameters after training (params).
753         - Loss history during training (numpy.ndarray).
754
755     """
756
757     sample_number = data.shape[0]
758
759     passes_per_epoch = sample_number // batch_size
760
761     loss_history = np.ones((max_epochs * passes_per_epoch, 1)) *
762         stop_loss
763
764     print('Starting training...')
765     print(f'Target Validation Loss: {stop_loss} | Maximum Number
766         of Epochs: {max_epochs}.\n')
767
768     epoch = 0
769     val_loss = stop_loss + 1
770
771     while val_loss > stop_loss and epoch < max_epochs:

```

```

765
766     dW_embed, dW1, dW2, db1, db2 = 0, 0, 0, 0, 0
767
768     perm = np.random.permutation(sample_number)
769     data = data[perm, :]
770     labels = labels[perm]
771
772     epoch += 1
773     epoch_loss = 0
774     correct_labels = 0
775
776     if verbose:
777         print(f'Epoch: [{epoch}/{max_epochs}] -> ', end='')
778
779     for batch in range(passes_per_epoch):
780
781         batch_data = data[batch*batch_size:(batch+1)*
782             batch_size, :]
783         batch_labels = labels[batch*batch_size:(batch+1)*
784             batch_size]
785
786         J, J_grad = nlpCost(params, batch_data, batch_labels,
787             dict_size)
788         A = NLP_forward_pass(params, batch_data, dict_size)['
789             A2']
790
791         epoch_loss += J
792         pred_labels = np.argmax(A, axis=1)
793         correct_labels += np.where(pred_labels ==
794             batch_labels, 1, 0).sum()
795
796         dW_embed = alpha * dW_embed - lr * J_grad['dW_embed']
797         dW1 = alpha * dW1 - lr * J_grad['dW1']
798         dW2 = alpha * dW2 - lr * J_grad['dW2']
799         db1 = alpha * db1 - lr * J_grad['db1']
800         db2 = alpha * db2 - lr * J_grad['db2']
801
802         params['W_embed'] += dW_embed
803         params['W1'] += dW1
804         params['W2'] += dW2
805         params['b1'] += db1
806         params['b2'] += db2
807
808         loss_history[(epoch-1)*passes_per_epoch + batch] = J
809
810     if sample_number % batch_size != 0:
811         batch_data = data[(passes_per_epoch*batch_size):, :]

```

```

811     batch_labels = labels[(passes_per_epoch*batch_size):]
812
813     J, J_grad = nlpCost(params, batch_data, batch_labels,
814                          dict_size)
815     A = NLP_forward_pass(params, batch_data, dict_size)['A2']
816
817     epoch_loss += J
818     pred_labels = np.argmax(A, axis=1)
819     correct_labels += np.where(pred_labels ==
820                               batch_labels, 1, 0).sum()
821
822     dW_embed = alpha * dW_embed - lr * J_grad['dW_embed']
823     dW1 = alpha * dW1 - lr * J_grad['dW1']
824     dW2 = alpha * dW2 - lr * J_grad['dW2']
825     db1 = alpha * db1 - lr * J_grad['db1']
826     db2 = alpha * db2 - lr * J_grad['db2']
827
828     params['W_embed'] += dW_embed
829     params['W1'] += dW1
830     params['W2'] += dW2
831     params['b1'] += db1
832     params['b2'] += db2
833
834     val_loss, _ = nlpCost(params, val_data, val_labels,
835                           dict_size)
836     val_A = NLP_forward_pass(params, val_data, dict_size)['A2']
837
838     val_preds = np.argmax(val_A, axis=1)
839     val_correct_labels = np.where(val_preds == val_labels, 1,
840                                   0).sum()
841     val_total_labels = val_data.shape[0]
842
843     train_acc_percent = (correct_labels / sample_number) *
844                          100
845     val_acc_percent = (val_correct_labels / val_total_labels)
846                      * 100
847     epoch_loss = epoch_loss * batch_size / sample_number
848
849     if verbose:
850         print(f'Training Loss: {epoch_loss:.4f} | Training
851               Accuracy: {train_acc_percent:.2f}% | Validation
852               Loss: {val_loss:.4f} | Validation Accuracy: {
853               val_acc_percent:.2f}%')
854
855     print('\nTraining completed.')
856
857     return [params, loss_history]

```

```

851
852
853 # The working code for the question-2
854 def q2():
855     with h5py.File('data2.h5', 'r') as File:
856         file_keys = list(File.keys())
857
858         testd_ind = np.array(File[file_keys[0]]) - 1
859         testx_ind = np.array(File[file_keys[1]]) - 1
860         traind_ind = np.array(File[file_keys[2]]) - 1
861         trainx_ind = np.array(File[file_keys[3]]) - 1
862         vald_ind = np.array(File[file_keys[4]]) - 1
863         valx_ind = np.array(File[file_keys[5]]) - 1
864         words = np.array(File[file_keys[6]])
865
866     num_words = words.shape[0]
867
868     # Initializing the
869     Wb_32_256 = init_NLP_Wb(num_words, 32, 256, 0.01)
870     Wb_16_128 = init_NLP_Wb(num_words, 16, 128, 0.01)
871     Wb_8_64 = init_NLP_Wb(num_words, 8, 64, 0.01)
872
873     max_epochs = 50
874
875     # Training for the instructed layer sizes
876     print('\nTraining for -> Embedding Size = 32 | Hidden Size =
877           256...\n')
878     Wb_32_256_trained, _ = SGD_Q2(Wb_32_256, num_words,
879                                   trainx_ind, traind_ind, valx_ind, vald_ind, 0.15, 200, 2.,
880                                   alpha=0.85, max_epochs=max_epochs, verbose=True)
881
882     print('\nTraining for -> Embedding Size = 16 | Hidden Size =
883           128...\n')
884     Wb_16_128_trained, _ = SGD_Q2(Wb_16_128, num_words,
885                                   trainx_ind, traind_ind, valx_ind, vald_ind, 0.15, 200, 2.,
886                                   alpha=0.85, max_epochs=max_epochs, verbose=True)
887
888     print('\nTraining for -> Embedding Size = 8 | Hidden Size =
889           64...\n')
890     Wb_8_64_trained, _ = SGD_Q2(Wb_8_64, num_words, trainx_ind,
891                                   traind_ind, valx_ind, vald_ind, 0.15, 200, 2., alpha=0.85,
892                                   max_epochs=max_epochs, verbose=True)
893
894     # Making predictions on the test data
895     pred_32_256 = NLP_forward_pass(Wb_32_256_trained, testx_ind,
896                                     num_words)['A2']
897     pred_16_128 = NLP_forward_pass(Wb_16_128_trained, testx_ind,
898                                     num_words)['A2']
899     pred_8_64 = NLP_forward_pass(Wb_8_64_trained, testx_ind,
900                                   num_words)['A2']

```

```

890
891 # Retriving the 10 most expected words
892 pred_32_256 = np.argsort(pred_32_256, axis=1)[: , -10:]
893 pred_16_128 = np.argsort(pred_16_128, axis=1)[: , -10:]
894 pred_8_64 = np.argsort(pred_8_64, axis=1)[: , -10:]
895
896 # Sampling 5 random trigrams from the test data set
897 test_size = testx_ind.shape[0]
898 number_of_samples = 5
899 random_samples = np.random.permutation(test_size)[:
    number_of_samples]
900
901 for i in range(number_of_samples):
902     print(f'\nTrigram: {words[testx_ind[random_samples[i]]]}
    |', f'True Value: {words[testd_ind[random_samples[i]
    ]]}')
903     print(f'Top Predictions for Embedding Size = 32, Hidden
    Size = 256: ', [word for word in reversed(words[
    pred_32_256[i]])])
904     print(f'Top Predictions for Embedding Size = 16, Hidden
    Size = 128: ', [word for word in reversed(words[
    pred_16_128[i]])])
905     print(f'Top Predictions for Embedding Size = 8, Hidden
    Size = 64: ', [word for word in reversed(words[
    pred_8_64[i]])], '\n')
906
907 #####
908 ##### CODE FOR Q-2 #####
909 #####
910
911 ## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION
912
913
914 def softmax(X, axis=1):
915     max_vals = np.max(X, axis=axis, keepdims=True)
916     e_X = np.exp(X - max_vals)
917     Y = e_X / np.sum(e_X, axis=axis, keepdims=True)
918     return Y
919
920
921 def tanh_backward(X):
922     return 1 - np.tanh(X)**2
923
924
925 def plot_confusion_matrix(predictions, true_labels, class_names=
    None):
926     """
927     Plots a confusion matrix for the given predictions and true
    labels without sklearn.
928
929     Parameters:

```

```

930 predictions (numpy.ndarray): Model predictions of shape (N,
    num_classes) (softmax outputs or similar).
931 true_labels (numpy.ndarray): True labels of shape (N,
    num_classes) (one-hot encoded).
932 class_names (array, optional): The names of the classes. If
    not provided, integers will be assigned.
933
934 Returns:
935 (matplotlib.figure.Figure): The confusion matrix plot.
936 """
937 # Convert one-hot encoded labels to integer labels
938 num_classes = true_labels.shape[1]
939 pred_classes = np.argmax(predictions, axis=1)
940 true_classes = np.argmax(true_labels, axis=1)
941
942 # Initialize confusion matrix
943 confusion_matrix = np.zeros((num_classes, num_classes), dtype
    =int)
944
945 # Populate confusion matrix
946 for t, p in zip(true_classes, pred_classes):
947     confusion_matrix[t, p] += 1
948
949 # Plot confusion matrix
950 plot = plt.figure(figsize=(10, 8))
951 plt.imshow(confusion_matrix, cmap='viridis')
952 plt.colorbar()
953 plt.xlabel("Predicted Class")
954 plt.ylabel("True Class")
955
956 if class_names == None:
957     plt.xticks(ticks=np.arange(num_classes), labels=[f"Class
    {i+1}" for i in range(num_classes)])
958     plt.yticks(ticks=np.arange(num_classes), labels=[f"Class
    {i+1}" for i in range(num_classes)])
959 else:
960     plt.xticks(ticks=np.arange(num_classes), labels=
    class_names)
961     plt.yticks(ticks=np.arange(num_classes), labels=
    class_names)
962
963 # Add numbers to each cell
964 for i in range(num_classes):
965     for j in range(num_classes):
966         value = confusion_matrix[i, j]
967         max_value = np.max(confusion_matrix) / 2
968
969         if value > max_value:
970             text_color = 'white'
971         else:

```



```

973         text_color = 'black'
974
975         plt.text(j, i, value, ha='center', va='center', color
976                 =text_color)
977
978     return plot
979
980
981
982 def initRecWb(input_size, hidden_size):
983     """ Initializes the layer parameters for simple recurrent
984     layer with Xavier initialization.
985
986     Args:
987         input_size (int): Number of input features.
988         hidden_size (int): Number of hidden neurons.
989
990     Returns:
991         dict: The dictionary of the recurrent layer weights.
992     """
993     w0_ih = np.sqrt(6 / (input_size + hidden_size))
994     W_ih = np.random.uniform(-w0_ih, w0_ih, (input_size,
995         hidden_size))
996
997     w0_hh = np.sqrt(6 / (2 * hidden_size))
998     W_hh = np.random.uniform(-w0_hh, w0_hh, (hidden_size,
999         hidden_size))
1000
1001     b_ih = np.zeros((1, hidden_size))
1002
1003     WbRec = {'W_hh': W_hh, 'W_ih': W_ih, 'b_ih': b_ih}
1004     return WbRec
1005
1006
1007 def initSeqWb(sizes):
1008     """ Initializes the layer parameters for MLP
1009     layer with Xavier initialization.
1010
1011     Args:
1012         sizes (array): the array of desired layer sizes.
1013
1014     Returns:
1015         dict: The dictionary of the layer weights for MLP.
1016     """
1017     Wb = {}
1018
1019     for i in range(len(sizes) - 1):
1020         w_0 = np.sqrt(6 / (sizes[i] + sizes[i+1]))
1021         Wb['W' + str(i)] = np.random.uniform(-w_0, w_0, (
1022             sizes[i], sizes[i+1]))
1023         Wb['b' + str(i)] = np.zeros((1, sizes[i + 1]))

```

```

1020
1021     return Wb
1022
1023 def reccurentForward(WbRec, data):
1024     """
1025     Performs a forward pass through a recurrent neural network
1026         layer.
1027
1028     Parameters
1029     -----
1030     WbRec : dict
1031         Dictionary containing the recurrent network weights and
1032         biases:
1033         - 'W_ih': Input-to-hidden weight matrix.
1034         - 'W_hh': Hidden-to-hidden weight matrix.
1035         - 'b_ih': Bias vector for hidden states.
1036     data : numpy.ndarray
1037         Input data of shape (batch_size, time_steps, features),
1038         where:
1039         - batch_size: Number of samples in a batch.
1040         - time_steps: Number of time steps in the sequence.
1041         - features: Number of features at each time step.
1042
1043     Returns
1044     -----
1045     list
1046         A list containing:
1047         - h_t: Hidden state of the last time step (numpy.ndarray)
1048             .
1049         - recCache: Dictionary containing intermediate
1050             calculations for backpropagation,
1051             including:
1052         - 'Zrec_<t>': Pre-activation values for each time step.
1053         - 'H_<t>': Hidden state for each time step.
1054
1055     Notes
1056     -----
1057     - Uses the hyperbolic tangent (tanh) activation function for
1058         the hidden states.
1059     - Outputs the final hidden state and a cache for all time
1060         steps.
1061     """
1062     batch_size, time_steps, features = data.shape
1063
1064     h_t = np.zeros((batch_size, WbRec['W_hh'].shape[0]))
1065
1066     recCache = {}
1067
1068     for t in range(time_steps):
1069         x_t = data[:, t, :]

```

```

1064         Z = np.matmul(x_t, WbRec['W_ih']) + np.matmul(h_t, WbRec[
1065             'W_hh']) + WbRec['b_ih']
1066         h_t = np.tanh(Z)
1067
1068         recCache['Zrec_' + str(t)] = Z
1069         recCache['H_' + str(t)] = h_t
1070
1071     return [h_t, recCache]
1072
1073 def sequentialForward(Wb, data):
1074     """
1075     Performs a forward pass through a sequential multi-layer
1076     feedforward neural network.
1077
1078     Parameters
1079     -----
1080     Wb : dict
1081         Dictionary containing the weights and biases for each
1082         layer:
1083         - 'W<i>': Weight matrix for layer i.
1084         - 'b<i>': Bias vector for layer i.
1085     data : numpy.ndarray
1086         Input data of shape (batch_size, input_features), where:
1087         - batch_size: Number of samples in a batch.
1088         - input_features: Number of input features.
1089
1090     Returns
1091     -----
1092     list
1093         A list containing:
1094         - A: Output of the final layer after the softmax
1095           activation (numpy.ndarray).
1096         - cache: Dictionary storing intermediate calculations for
1097           backpropagation, including:
1098         - 'A<i>': Activation output of layer i.
1099         - 'Z<i>': Pre-activation output of layer i.
1100
1101     Notes
1102     -----
1103     - Applies the tanh activation function for all layers except
1104       the last.
1105     - The last layer uses a softmax activation for multi-class
1106       classification.
1107     - Outputs the final activation and a cache for
1108       backpropagation.
1109     """
1110     num_layers = len(Wb) // 2
1111
1112     cache = {}

```

```

1107     A = data
1108
1109     cache['A-1'] = data
1110     cache['Z-1'] = np.zeros_like(data)
1111
1112     for layer in range(num_layers - 1):
1113         Z = np.matmul(A, Wb['W' + str(layer)]) + Wb['b' + str(
1114             layer)]
1115         A = np.tanh(Z)
1116
1117         cache['Z' + str(layer)] = Z
1118         cache['A' + str(layer)] = A
1119
1120     Z = np.matmul(A, Wb['W' + str(num_layers - 1)]) + Wb['b' +
1121         str(num_layers - 1)]
1122     A = softmax(Z, axis=1)
1123
1124     cache['Z' + str(num_layers - 1)] = Z
1125     cache['A' + str(num_layers - 1)] = A
1126
1127     return [A, cache]
1128
1129 def sequentialBackward(Wb, labels, cache):
1130     """
1131     Performs backpropagation through a sequential multi-layer
1132     feedforward neural network.
1133
1134     Parameters
1135     -----
1136     Wb : dict
1137         Dictionary containing the weights and biases for each
1138         layer:
1139         - 'W<i>': Weight matrix for layer i.
1140         - 'b<i>': Bias vector for layer i.
1141     labels : numpy.ndarray
1142         One-hot encoded labels of shape (batch_size, num_classes)
1143
1144     cache : dict
1145         Dictionary containing forward pass intermediate
1146         calculations, including:
1147         - 'A<i>': Activation output of layer i.
1148         - 'Z<i>': Pre-activation output of layer i.
1149
1150     Returns
1151     -----
1152     list
1153         A list containing:
1154         - J: Cross-entropy loss for the batch (float).
1155         - grads: Dictionary of gradients for weights and biases,
1156             including:

```

```

1151         - 'dW<i>': Gradient of weight matrix for layer i.
1152         - 'db<i>': Gradient of bias vector for layer i.
1153         - dA_prev (numpy.ndarray): Gradient of activation for
            inputing to an earlier layer's backpropagation.
1154
1155     Notes
1156     -----
1157     - Computes the cross-entropy loss for multi-class
            classification.
1158     - Gradients are calculated for all weights and biases in the
            network.
1159     - Assumes the use of 'tanh_backward' to compute the gradient
            of the tanh activation.
1160     """
1161
1162     batch_size = labels.shape[0]
1163
1164     num_layers = len(Wb) // 2
1165     # _, cache = sequentialForward(Wb, recLastState)
1166
1167     lastActivation = cache['A' + str(num_layers-1)]
1168
1169     J = -np.sum(labels * np.log(lastActivation)) / batch_size
1170     grads = {}
1171
1172     A_prev = cache['A' + str(num_layers-1)]
1173     dZ = lastActivation - labels
1174     dA_prev = 0
1175
1176     for layer in reversed(range(num_layers)):
1177         A_prev = cache['A' + str(layer-1)]
1178         grads['db' + str(layer)] = np.sum(dZ, axis=0, keepdims=
            True) / batch_size
1179         grads['dW' + str(layer)] = np.matmul(A_prev.T, dZ) /
            batch_size
1180         dA_prev = np.matmul(dZ, Wb['W' + str(layer)].T)
1181         dZ = dA_prev * tanh_backward(cache['Z' + str(layer-1)])
1182
1183     return [J, grads, dA_prev]
1184
1185 def recBackward(WbRec, input, last_dA, recCache,
1186 max_lookback_distance=None):
1187     """
1188     Backpropagation through time for a single-layer RNN.
1189
1190     Parameters
1191     -----
1192     WbRec : dict
1193         Dictionary containing the recurrent layer parameters:
1194         - 'W_ih': Input-to-hidden weights, shape (D_in, D_h)
1195         - 'W_hh': Hidden-to-hidden weights, shape (D_h, D_h)

```

```

1195         - 'b_ih': Bias for hidden state, shape (D_h,)
1196 last_dA : np.ndarray
1197     The gradient of the loss w.r.t. the last hidden state,
1198         shape (N, D_h).
1199 recCache : dict
1200     Cache from the forward pass containing:
1201     - 'X_t', 'Zrec_t', 'H_t' for t=1,...,T
1202     Must contain all time steps that were used in forward
1203     propagation.
1204 max_lookback_distance : int or None
1205     The number of steps to backprop through time.
1206     If None or greater than the total number of steps, all
1207     steps are used.
1208
1209 Returns
1210 -----
1211 grads : dict
1212     Dictionary containing:
1213     - 'dW_ih'
1214     - 'dW_hh'
1215     - 'db_ih'
1216 """
1217
1218 batch_size = input.shape[0]
1219
1220 W_ih = WbRec['W_ih']
1221 W_hh = WbRec['W_hh']
1222 b_ih = WbRec['b_ih']
1223
1224 # Extract the max t by checking keys:
1225 timesteps = [int(k.split('_')[1]) for k in recCache.keys() if
1226               k.startswith('H_')]
1227 T = max(timesteps) if len(timesteps) > 0 else 0
1228
1229 # If max_lookback_distance not provided or too large, use the
1230 # full length
1231 if max_lookback_distance is None or max_lookback_distance > T
1232 :
1233     max_lookback_distance = T
1234
1235 # Initialize gradients
1236 dW_ih = np.zeros_like(W_ih)
1237 dW_hh = np.zeros_like(W_hh)
1238 db_ih = np.zeros_like(b_ih)
1239
1240 # The hidden state gradient at the final step
1241 dH_next = last_dA
1242
1243 # Backprop through time
1244 for t in range(T, T - max_lookback_distance, -1):
1245     # Load cached values

```

```

1240     H_t = recCache[f'H_{t}']
1241     Z_t = recCache[f'Zrec_{t}']
1242     X_t = input[:, t, :]
1243
1244     # For H_{t-1}, if t=1, we might have H_0 in cache or use
1245     # zeros
1246     if t == 1:
1247         H_prev = recCache.get('H_0', np.zeros((X_t.shape[0],
1248             H_t.shape[1])))
1249     else:
1250         H_prev = recCache[f'H_{t-1}']
1251
1252     # Compute dZ_t
1253     dZ_t = dH_next * tanh_backward(Z_t)
1254
1255     # Compute gradients for parameters
1256     dW_ih += np.matmul(X_t.T, dZ_t) / batch_size
1257     dW_hh += np.matmul(H_prev.T, dZ_t) / batch_size
1258     db_ih += dZ_t.sum(axis=0) / batch_size # (D_h,)
1259
1260     dH_prev = np.matmul(dZ_t, W_hh.T)
1261
1262     # Update dH_next for the next iteration
1263     dH_next = dH_prev
1264
1265     recGrads = {
1266         'dW_ih': dW_ih,
1267         'dW_hh': dW_hh,
1268         'db_ih': db_ih
1269     }
1270
1271     return recGrads
1272
1273 #####
1274 ##### LSTM CODE #####
1275 #####
1276
1277 def initLSTMWb(input_size, hidden_size):
1278     WbLSTM = {}
1279     w0_ih = np.sqrt(6 / (input_size + hidden_size))
1280     w0_hh = np.sqrt(6 / (2 * hidden_size))
1281
1282     ih_names = ['W_f', 'W_i', 'W_o', 'W_c']
1283     hh_names = ['U_f', 'U_i', 'U_o', 'U_c']
1284     b_names = ['b_f', 'b_i', 'b_o', 'b_c']
1285
1286     WbLSTM['W_f'] = np.random.uniform(-w0_ih, w0_ih, (input_size,
1287         hidden_size)),
1288
1289     for name in ih_names:

```

```

1287         WbLSTM[name] = np.random.uniform(-w0_ih, w0_ih, (
1288             input_size, hidden_size))
1289     for name in hh_names:
1290         WbLSTM[name] = np.random.uniform(-w0_hh, w0_hh, (
1291             hidden_size, hidden_size))
1292     for name in b_names:
1293         WbLSTM[name] = np.zeros((1, hidden_size))
1294
1295     return WbLSTM
1296
1297 def lstmForward(WbLSTM, data):
1298     batch_size, time_steps, features = data.shape
1299
1300     W_f = WbLSTM['W_f']
1301     W_i = WbLSTM['W_i']
1302     W_o = WbLSTM['W_o']
1303     W_c = WbLSTM['W_c']
1304
1305     U_f = WbLSTM['U_f']
1306     U_i = WbLSTM['U_i']
1307     U_o = WbLSTM['U_o']
1308     U_c = WbLSTM['U_c']
1309
1310     b_f = WbLSTM['b_f']
1311     b_i = WbLSTM['b_i']
1312     b_o = WbLSTM['b_o']
1313     b_c = WbLSTM['b_c']
1314
1315     # Initialize hidden state (h_t) and cell state (c_t) to zeros
1316     hidden_size = W_f.shape[1]
1317     h_t = np.zeros((batch_size, hidden_size))
1318     c_t = np.zeros((batch_size, hidden_size))
1319
1320     lstmCache = {}
1321
1322     for t in range(time_steps):
1323         x_t = data[:, t, :]
1324
1325         # Compute gates
1326         f_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_f) + np.matmul(
1327             h_t, U_f) + b_f))))
1328         i_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_i) + np.matmul(
1329             h_t, U_i) + b_i))))
1330         o_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_o) + np.matmul(
1331             h_t, U_o) + b_o))))
1332         g_t = np.tanh(np.matmul(x_t, W_c) + np.matmul(h_t, U_c) +
1333             b_c)
1334
1335         # Update cell state and hidden state

```



```

1332         c_t = f_t * c_t + i_t * g_t
1333         h_t = o_t * np.tanh(c_t)
1334
1335         # Save intermediate values for backpropagation or
1336         # debugging
1337         lstmCache['f_t_'] + str(t)] = f_t
1338         lstmCache['i_t_'] + str(t)] = i_t
1339         lstmCache['o_t_'] + str(t)] = o_t
1340         lstmCache['g_t_'] + str(t)] = g_t
1341         lstmCache['c_t_'] + str(t)] = c_t
1342         lstmCache['h_t_'] + str(t)] = h_t
1343
1344     return [h_t, lstmCache]
1345
1346 def lstmBackward(WbLSTM, data, last_dA, lstmCache,
1347                 max_lookback_distance=None):
1348     """
1349     Backpropagation through time for an LSTM layer.
1350
1351     Parameters
1352     -----
1353     WbLSTM : dict
1354         Dictionary containing the LSTM layer parameters:
1355         - 'W_f', 'W_i', 'W_o', 'W_c': Input-to-hidden weights,
1356           each shape (D_in, D_h)
1357         - 'U_f', 'U_i', 'U_o', 'U_c': Hidden-to-hidden weights,
1358           each shape (D_h, D_h)
1359         - 'b_f', 'b_i', 'b_o', 'b_c': Biases for gates, each
1360           shape (D_h,)
1361     data : np.ndarray
1362         Input data to the LSTM, shape (N, T, D_in)
1363     last_dA : np.ndarray
1364         Gradient of the loss w.r.t. the last hidden state, shape
1365         (N, D_h)
1366     lstmCache : dict
1367         Cache from the forward pass containing:
1368         - 'f_t_t', 'i_t_t', 'o_t_t', 'g_t_t', 'c_t_t', 'h_t_t'
1369           for t=1,...,T
1370         Must contain all time steps that were used in forward
1371         propagation.
1372     max_lookback_distance : int or None
1373         The number of steps to backprop through time.
1374         If None or greater than the total number of steps, all
1375         steps are used.
1376
1377     Returns
1378     -----
1379     grads : dict
1380         Dictionary containing gradients for:
1381         - 'dW_f', 'dW_i', 'dW_o', 'dW_c'
1382         - 'dU_f', 'dU_i', 'dU_o', 'dU_c'

```

```

1374         - 'db_f', 'db_i', 'db_o', 'db_c'
1375     """
1376     N, T, D_in = data.shape
1377     hidden_size = WbLSTM['W_f'].shape[1]
1378
1379     # Initialize gradients for weights, biases, and recurrent
1380     # connections
1381     dW_f = np.zeros_like(WbLSTM['W_f'])
1382     dW_i = np.zeros_like(WbLSTM['W_i'])
1383     dW_o = np.zeros_like(WbLSTM['W_o'])
1384     dW_c = np.zeros_like(WbLSTM['W_c'])
1385
1386     dU_f = np.zeros_like(WbLSTM['U_f'])
1387     dU_i = np.zeros_like(WbLSTM['U_i'])
1388     dU_o = np.zeros_like(WbLSTM['U_o'])
1389     dU_c = np.zeros_like(WbLSTM['U_c'])
1390
1391     db_f = np.zeros_like(WbLSTM['b_f'])
1392     db_i = np.zeros_like(WbLSTM['b_i'])
1393     db_o = np.zeros_like(WbLSTM['b_o'])
1394     db_c = np.zeros_like(WbLSTM['b_c'])
1395
1396     # Initialize gradients w.r.t. hidden and cell states
1397     dH_next = last_dA
1398     dC_next = np.zeros((N, hidden_size))
1399
1400     # If max_lookback_distance is not provided, use the full
1401     # sequence length
1402     if max_lookback_distance is None or max_lookback_distance > T
1403         :
1404         max_lookback_distance = T
1405
1406     # Backprop through time
1407     for t in range(T - 1, T - max_lookback_distance - 1, -1):
1408         # Load cached values for time step t
1409         f_t = lstmCache[f'f_t_{t}']
1410         i_t = lstmCache[f'i_t_{t}']
1411         o_t = lstmCache[f'o_t_{t}']
1412         g_t = lstmCache[f'g_t_{t}']
1413         c_t = lstmCache[f'c_t_{t}']
1414         h_t = lstmCache[f'h_t_{t}']
1415         x_t = data[:, t, :]
1416         h_prev = lstmCache[f'h_t_{t-1}'] if t > 0 else np.
1417             zeros_like(h_t)
1418         c_prev = lstmCache[f'c_t_{t-1}'] if t > 0 else np.
1419             zeros_like(c_t)
1420
1421         # Gradients w.r.t. cell state and output gate
1422         dO_t = dH_next * np.tanh(c_t) # Gradient of output gate
1423         dC_t = dH_next * o_t * (1 - np.tanh(c_t)**2) + dC_next #
1424             Gradient of cell state

```

```

1419
1420     # Gradients w.r.t. gates
1421     dF_t = dC_t * c_prev
1422     dI_t = dC_t * g_t
1423     dG_t = dC_t * i_t
1424
1425     # Apply activation function derivatives
1426     dF_t *= f_t * (1 - f_t) # Sigmoid derivative
1427     dI_t *= i_t * (1 - i_t) # Sigmoid derivative
1428     dO_t *= o_t * (1 - o_t) # Sigmoid derivative
1429     dG_t *= 1 - g_t**2      # Tanh derivative
1430
1431     # Accumulate parameter gradients
1432     dW_f += np.matmul(x_t.T, dF_t) / N
1433     dW_i += np.matmul(x_t.T, dI_t) / N
1434     dW_o += np.matmul(x_t.T, dO_t) / N
1435     dW_c += np.matmul(x_t.T, dG_t) / N
1436
1437     dU_f += np.matmul(h_prev.T, dF_t) / N
1438     dU_i += np.matmul(h_prev.T, dI_t) / N
1439     dU_o += np.matmul(h_prev.T, dO_t) / N
1440     dU_c += np.matmul(h_prev.T, dG_t) / N
1441
1442     db_f += dF_t.sum(axis=0) / N
1443     db_i += dI_t.sum(axis=0) / N
1444     db_o += dO_t.sum(axis=0) / N
1445     db_c += dG_t.sum(axis=0) / N
1446
1447     # Backpropagate into previous hidden and cell states
1448     dH_next = np.matmul(dF_t, WbLSTM['U_f'].T) + \
1449                 np.matmul(dI_t, WbLSTM['U_i'].T) + \
1450                 np.matmul(dO_t, WbLSTM['U_o'].T) + \
1451                 np.matmul(dG_t, WbLSTM['U_c'].T)
1452
1453     dC_next = dC_t * f_t
1454
1455     # Package gradients into a dictionary
1456     lstmGrads = {
1457         'dW_f': dW_f, 'dW_i': dW_i, 'dW_o': dW_o, 'dW_c': dW_c,
1458         'dU_f': dU_f, 'dU_i': dU_i, 'dU_o': dU_o, 'dU_c': dU_c,
1459         'db_f': db_f, 'db_i': db_i, 'db_o': db_o, 'db_c': db_c
1460     }
1461
1462     return lstmGrads
1463
1464
1465 #####
1466 # GRU CODE
1467 #####
1468
1469 def initGRUWb(input_size, hidden_size):

```

```

1470 WbGRU = {}
1471
1472 w0_hh = np.sqrt(6 / (2 * hidden_size))
1473 w0_ih = np.sqrt(6 / (input_size + hidden_size))
1474
1475 WbGRU['W_z'] = np.random.uniform(-w0_ih, w0_ih, (input_size,
1476 hidden_size))
1477 WbGRU['W_r'] = np.random.uniform(-w0_ih, w0_ih, (input_size,
1478 hidden_size))
1479 WbGRU['W_h'] = np.random.uniform(-w0_ih, w0_ih, (input_size,
1480 hidden_size))
1481
1482 WbGRU['U_z'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size,
1483 hidden_size))
1484 WbGRU['U_r'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size,
1485 hidden_size))
1486 WbGRU['U_h'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size,
1487 hidden_size))
1488
1489 WbGRU['b_z'] = np.zeros((1, hidden_size))
1490 WbGRU['b_r'] = np.zeros((1, hidden_size))
1491 WbGRU['b_h'] = np.zeros((1, hidden_size))
1492
1493 return WbGRU
1494
1495 def gruForward(WbGRU, data):
1496     batch_size, time_steps, features = data.shape
1497
1498     W_z = WbGRU['W_z']
1499     W_r = WbGRU['W_r']
1500     W_h = WbGRU['W_h']
1501
1502     U_z = WbGRU['U_z']
1503     U_r = WbGRU['U_r']
1504     U_h = WbGRU['U_h']
1505
1506     b_z = WbGRU['b_z']
1507     b_r = WbGRU['b_r']
1508     b_h = WbGRU['b_h']
1509
1510     # Initialize hidden state (h_t) to zeros
1511     hidden_size = W_z.shape[1]
1512     h_t = np.zeros((batch_size, hidden_size))
1513
1514     gruCache = {}
1515
1516     for t in range(time_steps):
1517         x_t = data[:, t, :]
1518
1519         # Compute gates

```

```

1514     z_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_z) + np.matmul(
1515         h_t, U_z) + b_z)))
1516     r_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_r) + np.matmul(
1517         h_t, U_r) + b_r)))
1518     h_hat_t = np.tanh(np.matmul(x_t, W_h) + np.matmul(r_t *
1519         h_t, U_h) + b_h)
1520
1521     # Update hidden state
1522     h_t = z_t * h_t + (1 - z_t) * h_hat_t
1523
1524     # Save intermediate values for backpropagation or
1525     debugging
1526     gruCache['z_t_'] + str(t)] = z_t
1527     gruCache['r_t_'] + str(t)] = r_t
1528     gruCache['h_hat_t_'] + str(t)] = h_hat_t
1529     gruCache['h_t_'] + str(t)] = h_t
1530
1531     return [h_t, gruCache]
1532
1533 def gruBackward(WbGRU, data, last_dA, gruCache,
1534     max_lookback_distance=None):
1535     """
1536     Backpropagation through time for a GRU layer.
1537
1538     Parameters
1539     -----
1540     WbGRU : dict
1541         Dictionary containing the GRU layer parameters:
1542         - 'W_z', 'W_r', 'W_h': Input-to-hidden weights, each
1543           shape (D_in, D_h)
1544         - 'U_z', 'U_r', 'U_h': Hidden-to-hidden weights, each
1545           shape (D_h, D_h)
1546         - 'b_z', 'b_r', 'b_h': Biases for gates, each shape (
1547           D_h,)
1548     data : np.ndarray
1549         Input data to the GRU, shape (N, T, D_in)
1550     last_dA : np.ndarray
1551         Gradient of the loss w.r.t. the last hidden state, shape
1552         (N, D_h)
1553     gruCache : dict
1554         Cache from the forward pass containing:
1555         - 'z_t_t', 'r_t_t', 'h_hat_t_t', 'h_t_t' for t=1,...,T
1556         Must contain all time steps that were used in forward
1557         propagation.
1558     max_lookback_distance : int or None
1559         The number of steps to backprop through time.
1560         If None or greater than the total number of steps, all
1561         steps are used.
1562
1563     Returns

```

```

1554 -----
1555 grads : dict
1556     Dictionary containing gradients for:
1557         - 'dW_z', 'dW_r', 'dW_h'
1558         - 'dU_z', 'dU_r', 'dU_h'
1559         - 'db_z', 'db_r', 'db_h'
1560     """
1561 N, T, D_in = data.shape
1562 hidden_size = WbGRU['W_z'].shape[1]
1563
1564 # Initialize gradients for weights, biases, and recurrent
1565     connections
1566 dW_z = np.zeros_like(WbGRU['W_z'])
1567 dW_r = np.zeros_like(WbGRU['W_r'])
1568 dW_h = np.zeros_like(WbGRU['W_h'])
1569
1570 dU_z = np.zeros_like(WbGRU['U_z'])
1571 dU_r = np.zeros_like(WbGRU['U_r'])
1572 dU_h = np.zeros_like(WbGRU['U_h'])
1573
1574 db_z = np.zeros_like(WbGRU['b_z'])
1575 db_r = np.zeros_like(WbGRU['b_r'])
1576 db_h = np.zeros_like(WbGRU['b_h'])
1577
1578 # Initialize gradients w.r.t. hidden state
1579 dH_next = last_dA
1580
1581 # If max_lookback_distance is not provided, use the full
1582     sequence length
1583 if max_lookback_distance is None or max_lookback_distance > T
1584     :
1585     max_lookback_distance = T
1586
1587 # Backprop through time
1588 for t in range(T - 1, T - max_lookback_distance - 1, -1):
1589     z_t = gruCache[f'z_t_{t}']
1590     r_t = gruCache[f'r_t_{t}']
1591     h_hat_t = gruCache[f'h_hat_t_{t}']
1592     h_t = gruCache[f'h_t_{t}']
1593     x_t = data[:, t, :]
1594     h_prev = gruCache[f'h_t_{t-1}'] if t > 0 else np.
1595         zeros_like(h_t)
1596
1597     # Gradients w.r.t. hidden state
1598     dZ_t = dH_next * (h_prev - h_hat_t)
1599     dH_hat_t = dH_next * (1 - z_t)
1600     dH_prev = dH_next * z_t
1601
1602     # Apply activation function derivatives
1603     dZ_t *= z_t * (1 - z_t) # Sigmoid derivative
1604     dR_t = np.matmul(dH_hat_t, WbGRU['U_h'].T) * h_prev

```

```

1601         dR_t *= r_t * (1 - r_t) # Sigmoid derivative
1602         dH_hat_t *= 1 - h_hat_t**2 # Tanh derivative
1603
1604         # Accumulate parameter gradients
1605         dW_z += np.matmul(x_t.T, dZ_t) / N
1606         dW_r += np.matmul(x_t.T, dR_t) / N
1607         dW_h += np.matmul(x_t.T, dH_hat_t) / N
1608
1609         dU_z += np.matmul(h_prev.T, dZ_t) / N
1610         dU_r += np.matmul(h_prev.T, dR_t) / N
1611         dU_h += np.matmul((r_t * h_prev).T, dH_hat_t) / N
1612
1613         db_z += dZ_t.sum(axis=0) / N
1614         db_r += dR_t.sum(axis=0) / N
1615         db_h += dH_hat_t.sum(axis=0) / N
1616
1617         # Backpropagate into previous hidden state
1618         dH_next = dH_prev + np.matmul(dZ_t, WbGRU['U_z'].T) + np.
            matmul(dR_t, WbGRU['U_r'].T)
1619
1620     gruGrads = {
1621         'dW_z': dW_z, 'dW_r': dW_r, 'dW_h': dW_h,
1622         'dU_z': dU_z, 'dU_r': dU_r, 'dU_h': dU_h,
1623         'db_z': db_z, 'db_r': db_r, 'db_h': db_h
1624     }
1625
1626     return gruGrads
1627
1628
1629 #####
1630 ### SGD CODE AND UTILS
1631 #####
1632
1633
1634 def SGD_Q3(WbRec, Wb, data, labels, val_data, val_labels, lr,
            batch_size,
1635             stop_loss=0.1, max_epochs=50, alpha=0, verbose=True,
            recurrent_type='Simple', max_lookback_distance=None
            ):
1636
1637
1638     sample_number = data.shape[0]
1639
1640     recFuncForward = reccurentForward
1641     recFuncBackward = recBackward
1642
1643     # Determine the typ of recurrent layer.
1644     if recurrent_type == 'LSTM' or recurrent_type == 'lstm':
1645         recFuncForward = lstmForward
1646         recFuncBackward = lstmBackward
1647     elif recurrent_type == 'GRU' or recurrent_type == 'gru':

```

```

1648         recFuncForward = gruForward
1649         recFuncBackward = gruBackward
1650
1651     # Inicializa the dictionary that holds gradients.
1652     dParams = {}
1653     for param in Wb:
1654         dParams['d' + param] = np.zeros_like(Wb[param])
1655     for recParam in WbRec:
1656         dParams['d' + recParam] = np.zeros_like(WbRec[recParam])
1657
1658
1659     passes_per_epoch = sample_number // batch_size
1660
1661     # Training metrics arrays
1662     train_loss_history = np.zeros((max_epochs, 1))
1663     val_loss_history = np.zeros((max_epochs, 1))
1664     train_accuracy_history = np.zeros((max_epochs, 1))
1665     val_accuracy_history = np.zeros((max_epochs, 1))
1666
1667     J_train = 0
1668     J_val = 0
1669     val_acc_percent = 0
1670     train_acc_percent = 0
1671
1672     print('Starting training...')
1673     print('Recurrency type:', recurrent_type)
1674     print(f'Target Validation Loss: {stop_loss} | Maximum Number
        of Epochs: {max_epochs}.\n')
1675
1676     epoch = 0
1677     J_val = stop_loss + 1
1678
1679     # Starting training
1680     while J_val > stop_loss and epoch != max_epochs:
1681
1682         # Taking a random permutations of the data and the labels
1683         .
1684         perm = np.random.permutation(sample_number)
1685         data = data[perm, :]
1686         labels = labels[perm, :]
1687
1688         J_train = 0
1689         J_val = 0
1690
1691         correct_labels = 0
1692         total_labels = sample_number
1693         epoch += 1
1694
1695         if verbose:
1696             print(f'Epoch: [{epoch}/{max_epochs}] -> ', end='')

```



```

1697 # Mini-batching
1698 for batch in range(passes_per_epoch):
1699
1700     batch_data = batch*batch_size
1701
1702     # Forward propagation through the network
1703     h_t, recCache = recFuncForward(WbRec, data[batch_data
1704                                     :batch_data + batch_size, :])
1705     A, cache = sequentialForward(Wb, h_t)
1706
1707     # Cross-entropy cost calculation.
1708     J, grads, last_dA = sequentialBackward(Wb, labels[
1709         batch_data:batch_data + batch_size, :], cache)
1710     recGrads = recFuncBackward(WbRec, data[batch_data:
1711         batch_data + batch_size, :], last_dA, recCache,
1712         max_lookback_distance)
1713
1714     # Calculating accuracy
1715     J_train += J
1716     pred_labels = np.argmax(A, axis=1)
1717     true_labels = np.argmax(labels[batch_data:batch_data
1718         + batch_size, :], axis=1)
1719     correct_labels += np.where(pred_labels == true_labels
1720         , 1, 0).sum()
1721
1722     # Applying the momentum term
1723     for d_param in grads:
1724         dParams[d_param] = alpha * dParams[d_param] - lr
1725         * grads[d_param]
1726     for d_param in recGrads:
1727         dParams[d_param] = alpha * dParams[d_param] - lr
1728         * recGrads[d_param]
1729
1730     # Updating the weights and biases
1731     for param in Wb:
1732         Wb[param] += dParams['d' + param]
1733     for param in WbRec:
1734         WbRec[param] += dParams['d' + param]
1735
1736     # Doing the same steps for the residual batch.
1737     if sample_number % batch_size != 0:
1738
1739         batch_data = passes_per_epoch*batch_size
1740
1741         h_t, recCache = recFuncForward(WbRec, data[batch_data
1742             :, :])
1743         A, cache = sequentialForward(Wb, h_t)
1744
1745         J, grads, last_dA = sequentialBackward(Wb, labels[
1746             batch_data:, :], cache)

```

```

1738         recGrads = recFuncBackward(WbRec, data[batch_data:,
1739                                     :], last_dA, recCache, max_lookback_distance)
1740
1741     J_train += J
1742     pred_labels = np.argmax(A, axis=1)
1743     true_labels = np.argmax(labels[batch_data:batch_data
1744                                   + batch_size, :], axis=1)
1745     correct_labels += np.where(pred_labels == true_labels
1746                                , 1, 0).sum()
1747
1748     for d_param in grads:
1749         dParams[d_param] = alpha * dParams[d_param] - lr
1750         * grads[d_param]
1751     for d_param in recGrads:
1752         dParams[d_param] = alpha * dParams[d_param] - lr
1753         * recGrads[d_param]
1754
1755     for param in Wb:
1756         Wb[param] += dParams['d' + param]
1757     for param in WbRec:
1758         WbRec[param] += dParams['d' + param]
1759
1760     h_t, recCache = recFuncForward(WbRec, val_data)
1761     A, cache = sequentialForward(Wb, h_t)
1762
1763     # Validation data set calculations
1764     J_val, _, _ = sequentialBackward(Wb, val_labels, cache)
1765     val_pred_labels = np.argmax(A, axis=1)
1766     val_true_labels = np.argmax(val_labels, axis=1)
1767     val_correct_labels = np.where(val_pred_labels ==
1768                                   val_true_labels, 1, 0).sum()
1769     val_total_labels = val_labels.shape[0]
1770
1771     train_acc_percent = (correct_labels / total_labels) * 100
1772     val_acc_percent = (val_correct_labels / val_total_labels)
1773     * 100
1774     J_train = J_train * batch_size / sample_number
1775
1776     # Updating the metrics arrays.
1777     train_loss_history[epoch-1] = J_train
1778     val_loss_history[epoch-1] = J_val
1779     train_accuracy_history[epoch-1] = train_acc_percent
1780     val_accuracy_history[epoch-1] = val_acc_percent
1781
1782     if verbose:
1783         print(f'Training Cost: {J_train:.4f} | Training
1784               Accuracy: {train_acc_percent:.2f}% | Validation
1785               Cost: {J_val:.4f} | Validation Accuracy: {

```

```

1780         val_acc_percent:.2f}%')
1781
1782     train_loss_history[epoch:] = J_train
1783     val_loss_history[epoch:] = J_val
1784     train_accuracy_history[epoch:] = train_acc_percent
1785     val_accuracy_history[epoch:] = val_acc_percent
1786
1787     metrics = {'train_loss':train_loss_history, 'val_loss':
1788               val_loss_history,
1789               'train_accuracy':train_accuracy_history, '
1790               val_accuracy':val_accuracy_history}
1791
1792     print('\nTraining completed.')
1793
1794     return [WbRec, Wb, metrics]
1795
1796 def get_accuracy(WbRec, Wb, testing_data, testing_labels,
1797                 recurrency_type='Simple'):
1798
1799     Forward = reccurentForward
1800
1801     if recurrency_type == 'Simple' or recurrency_type == 'simple':
1802         :
1803         Forward = reccurentForward
1804     elif recurrency_type == 'LSTM' or recurrency_type == 'lstm':
1805         Forward = lstmForward
1806     elif recurrency_type == 'GRU' or recurrency_type == 'gru':
1807         Forward = gruForward
1808
1809     h_t, recCache = Forward(WbRec, testing_data)
1810     A, cache = sequantialForward(Wb, h_t)
1811
1812     pred_labels = np.argmax(A, axis=1)
1813     true_labels = np.argmax(testing_labels, axis=1)
1814
1815     correct = np.sum(np.where(true_labels == pred_labels, 1, 0))
1816     total = testing_data.shape[0]
1817
1818     return [correct / total, A]
1819
1820 def initWbQ3(layer_sizes, recurrency_type):
1821     """ Initializes the network for the desired recurrency type
1822         and layer sizes
1823
1824     Args:
1825         layer_sizes (_type_): _description_
1826         recurrency_type (array): The array of layer sizes.
1827
1828     Returns:

```

```

1825         list: list containing the weights and biases:
1826         WbRec (dict): Recurrent layer parameters.
1827         Wb (dict): MLP layer parameters.
1828     """
1829     if recurrency_type == 'Simple' or recurrency_type == 'simple':
1830         :
1831         initFuncWb = initRecWb
1832     elif recurrency_type == 'LSTM' or recurrency_type == 'lstm':
1833         initFuncWb = initLSTMWb
1834     elif recurrency_type == 'GRU' or recurrency_type == 'gru':
1835         initFuncWb = initGRUWb
1836
1837     WbRec = initFuncWb(layer_sizes[0], layer_sizes[1])
1838     WbSeq = initSeqWb(layer_sizes[1:])
1839
1840     return [WbRec, WbSeq]
1841
1842 def q3():
1843     """ The working code for question-3. Takes and returns no
1844         arguments.
1845     """
1846     with h5py.File('data3.h5', 'r') as file:
1847         trX = np.array(file['trX'])
1848         trY = np.array(file['trY'])
1849         tstX = np.array(file['tstX'])
1850         tstY = np.array(file['tstY'])
1851
1852     # data preprocessing to normalize
1853     data_mean = trX.mean()
1854     data_std = trX.std()
1855
1856     trX = (trX - data_mean) / data_std
1857     tstX = (tstX - data_mean) / data_std
1858
1859     data_count = trX.shape[0]
1860
1861     # train / validation splitting the data
1862     val_train_split = 0.1
1863     train_start_index = int(data_count * val_train_split)
1864
1865     rand_perm = np.random.permutation(data_count)
1866     trX = trX[rand_perm]
1867     trY = trY[rand_perm]
1868
1869     data_train = trX[train_start_index:]
1870     data_val = trX[:train_start_index]
1871
1872     labels_train = trY[train_start_index:]
1873     labels_val = trY[:train_start_index]

```

```

1874
1875 print('Training data count:', data_train.shape[0])
1876 print('Validation data count:', data_val.shape[0], '\n')
1877
1878 recurrent_types = ['Simple', 'LSTM', 'GRU']
1879 max_lookback_list = {'Simple':None, 'LSTM':None, 'GRU':None}
1880 layer_sizes = [3, 128, 64, 64, 6]
1881 class_names = ['downstairs', 'jogging', 'sitting', 'standing',
1882               , 'upstairs', 'walking']
1883
1884 # Model hyperparameters
1885 learning_rate = 0.01
1886 batch_size = 32
1887 max_epochs = 50
1888 stop_loss = 0.5
1889 alpha = 0.85
1890 epochs = np.arange(1, max_epochs+1, 1)
1891
1892 for rec_type in recurrent_types:
1893     # initializing weights and biases
1894     WbRec, Wb = initWbQ3(layer_sizes, rec_type)
1895
1896     # training the network
1897     trainedWbRec, trainedWb, loss_hist = SGD_Q3(WbRec, Wb,
1898                                                data_train,
1899                                                labels_train,
1900                                                data_val, labels_val,
1901                                                lr=learning_rate,
1902                                                batch_size=
1903                                                    batch_size,
1904                                                    stop_loss=
1905                                                        stop_loss,
1906                                                    max_epochs=max_epochs
1907                                                        , alpha=alpha,
1908                                                        max_lookback_distance
1909                                                            =max_lookback_list
1910                                                            [rec_type],
1911                                                        recurrent_type=
1912                                                            rec_type)
1913
1914     # Displaying and saving the results
1915     print(f'\nCreating confusion matrix for {rec_type}
1916           recurrent cell test set...')
1917     test_accuracy, test_predictions = get_accuracy(
1918         trainedWbRec, trainedWb, tstX, tstY, rec_type)
1919     conf_mat = plot_confusion_matrix(test_predictions, tstY,
1920                                     class_names)
1921     plt.title(F'Confusion Matrix Over Test Set for {rec_type}
1922              Recurrent Cell')
1923     plt.savefig(f'conf_mat_{rec_type}_test.png')

```

```

1911
1912     print(f'\nCreating confusion matrix for {rec_type}
1913           recurrent cell training set...')
1914     train_accuracy, train_predictions = get_accuracy(
1915         trainedWbRec, trainedWb, data_train, labels_train,
1916         rec_type)
1917     conf_mat = plot_confusion_matrix(train_predictions,
1918         labels_train, class_names)
1919     plt.title(F'Confusion Matrix Over Training Set for {
1920         rec_type} Recurrent Cell')
1921     plt.savefig(f'conf_mat_{rec_type}_train.png')
1922
1923     print(f'\nCreating the training costs and accuracy plot
1924           for {rec_type} recurrent cell...')
1925     metrics_figure = plt.figure(figsize=(15,10))
1926
1927     metrics_figure.add_subplot(2,2,1)
1928     plt.plot(epochs, loss_hist['train_loss'], 'r')
1929     plt.xlabel('Epochs')
1930     plt.ylabel('Training Cost')
1931
1932     metrics_figure.add_subplot(2,2,2)
1933     plt.plot(epochs, loss_hist['val_loss'], 'r')
1934     plt.xlabel('Epochs')
1935     plt.ylabel('Validadtion Cost')
1936
1937     metrics_figure.add_subplot(2,2,3)
1938     plt.plot(epochs, loss_hist['train_accuracy'], 'b')
1939     plt.xlabel('Epochs')
1940     plt.ylabel('Training Accuracy (%)')
1941
1942     metrics_figure.add_subplot(2,2,4)
1943     plt.plot(epochs, loss_hist['val_accuracy'], 'b')
1944     plt.xlabel('Epochs')
1945     plt.ylabel('Validation Accuracy (%)')
1946
1947     plt.suptitle(f"Training Metrics for {rec_type} Recurrent
1948         Cell\nTest Accuracy: {100 * test_accuracy:.2f}%")
1949     plt.savefig(f'loss_and_acc_{rec_type}')
1950
1951     print(f'\nTest accuracy for {rec_type} recurrent cell:
1952         {100 * test_accuracy:.2f}%\n')
1953
1954     plt.show()
1955
1956 import sys
1957
1958 question = sys.argv[1]

```

```
1954
1955 def ozan_cem_bas_22102757_mini_project(question):
1956     if question == '1':
1957         q1()
1958     elif question == '2':
1959         q2()
1960     elif question == '3':
1961         q3()
1962     else:
1963         print("Please enter '1', '2' or '3' in argv to run the
1964               corresponding questions program.")
1965
1966 ozan_cem_bas_22102757_mini_project(question)
```