EEE 443/543: Neural Networks
Mini Project Report

Name: Ozan Cem Baş
ID: 22102757
Date: December 22, 2024

December 22, 2024

Question-1.

1.a) Preprocessing The Images

Figure 1: 200 randomly chosen colored images.

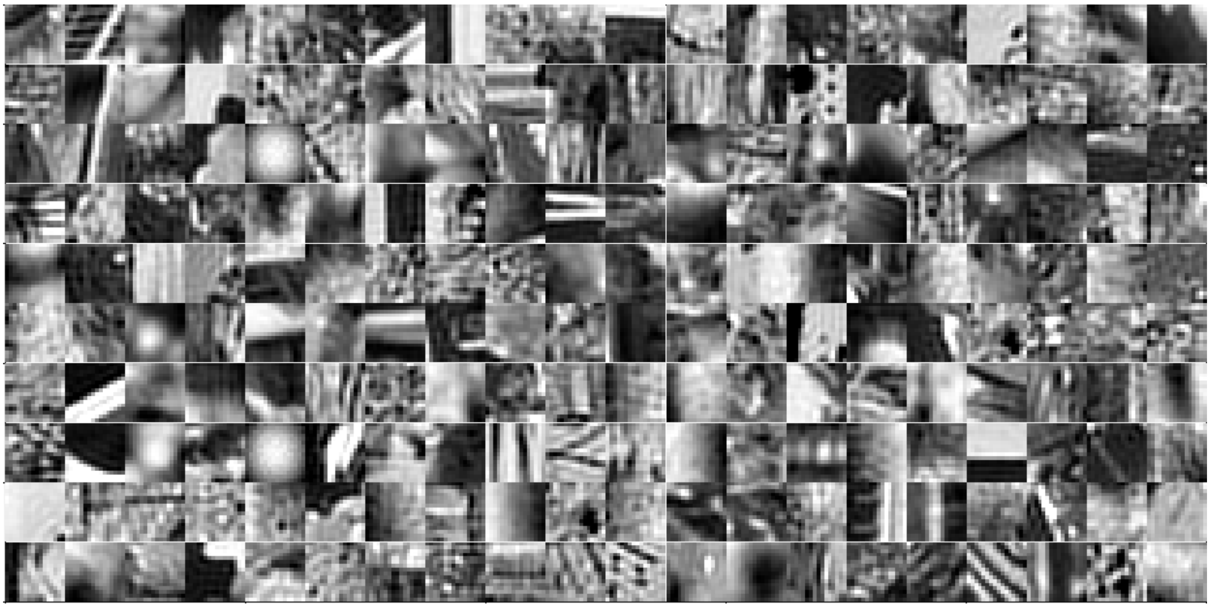


Figure 2: 200 randomly chosen images after converting to gray-scale and normalizing.

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 biasses $b^{(1)}$ and $b^{(2)}$ by randomly sampling their elements from a uniform distribution from the internal $[-w_0, w_0]$. Here,

$$w_0 = \sqrt{\frac{6}{L_{pre} + L_{post}}} \tag{1}$$

  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:

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

- 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:

$$\beta = 0.1$$
$$\rho = 0.5$$
$$\text{Batch Size} = 32$$
$$\text{epochs} = 20$$

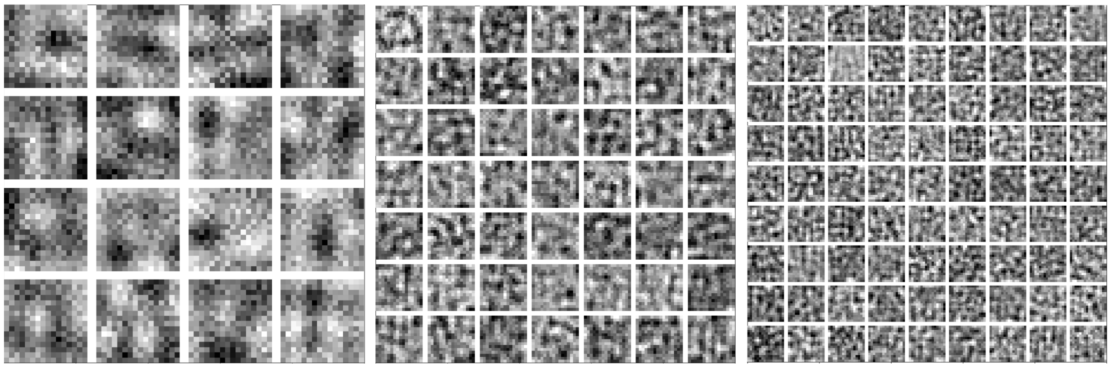
Please note that the parameters of the Adam optimizer are taken as $\beta_1 = 0.9$, $\beta_2 = 0.99$ and these values do not have a correlation to β in the loss function.

1.c) Hidden Layer Weights

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

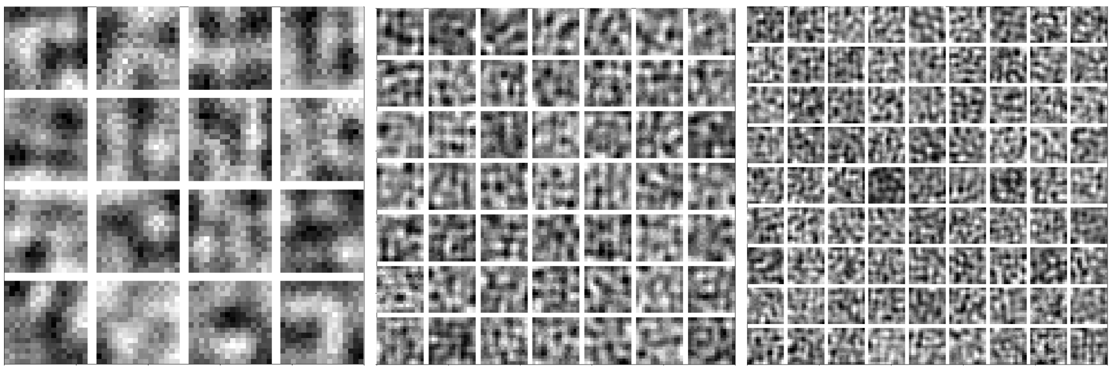
In the figure 3, the connection weights for each neuron in the hidden layer can be seen. The bright areas for each neurons' connection weights display how much their activation depends on the different parts of the image. It can be seen that some neurons are more susceptible to different parts of the image. Some neurons are seen to be more sensitive to circular shapes in the images while some ae 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



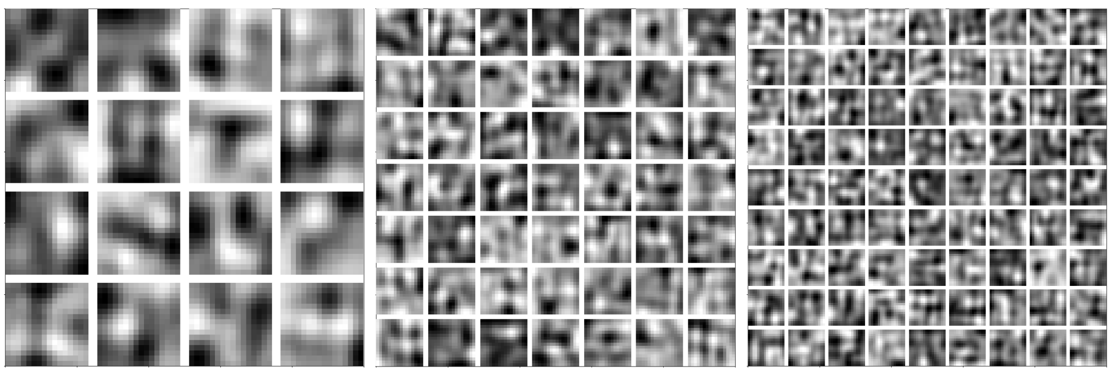
(a) $L_{hid} = 16$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

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



(a) $L_{hid} = 16$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

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



(a) $L_{hid} = 16$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

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,

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

- 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,

$$\begin{aligned} L_{embed} &= 32 \\ L_{hidden} &= 256 \\ \text{Learning Rate} &= 0.15 \\ \text{Batch Size} &= 200 \\ \alpha &= 0.85 \end{aligned}$$

.

- 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']` $\longrightarrow$ **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']` $\longrightarrow$ **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']` $\longrightarrow$ **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']` $\longrightarrow$ **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']` $\longrightarrow$ **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 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,
$$w_0 = \sqrt{\frac{6}{L_{pre} + L_{post}}} \tag{3}$$
- 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: *reccurent-Forward* 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:
$$\Delta w_{ij} = -\eta \frac{\partial J_{total}}{\partial w_{ij}} = -\eta \sum_{t=t_0}^{t_{end}} \frac{\partial J(t)}{\partial w_{ij}} \tag{4}$$
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:

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

  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,

$$W(n) = W(n-1) + \Delta W(N) \tag{6}$$

- *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.

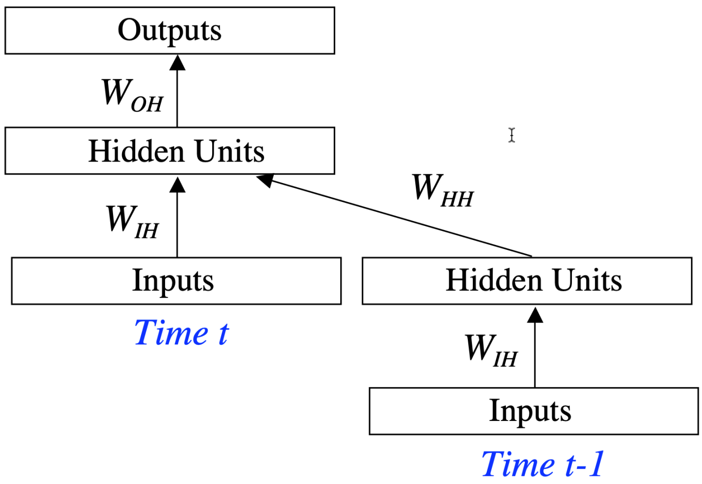
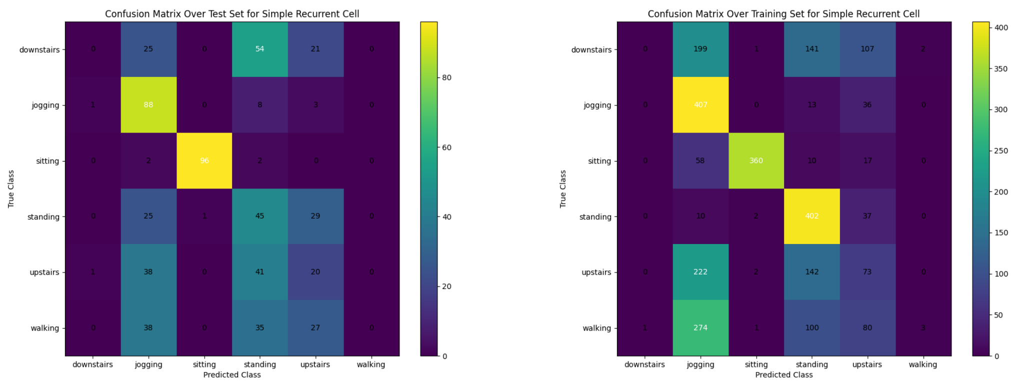


Figure 7: Simple recurrent neural network unfolded in time.

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

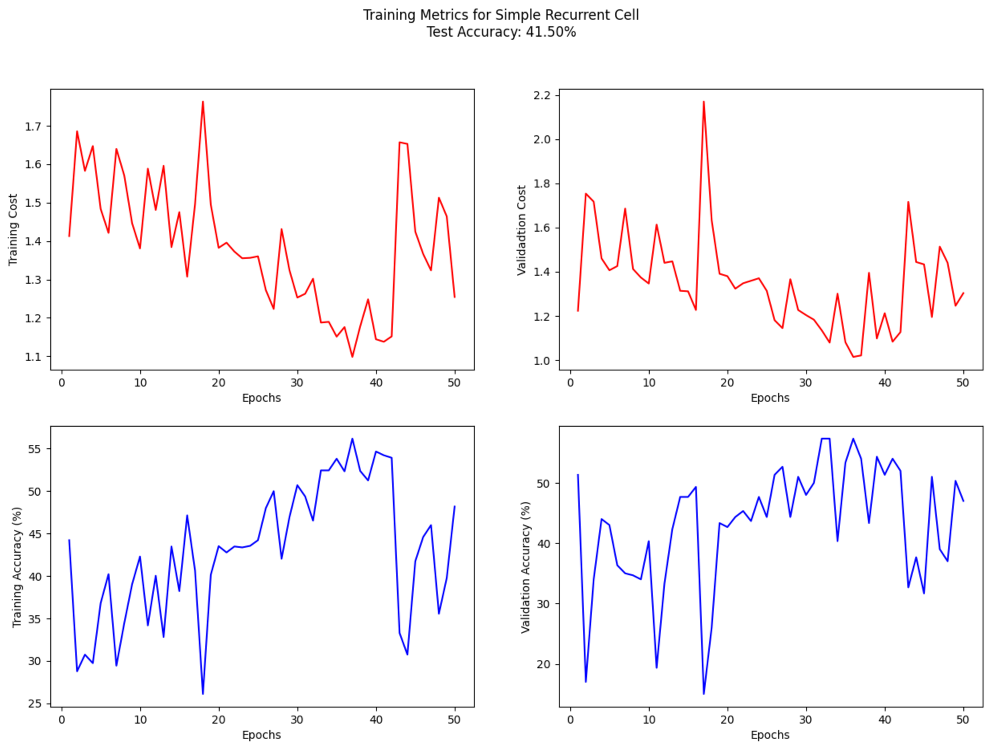
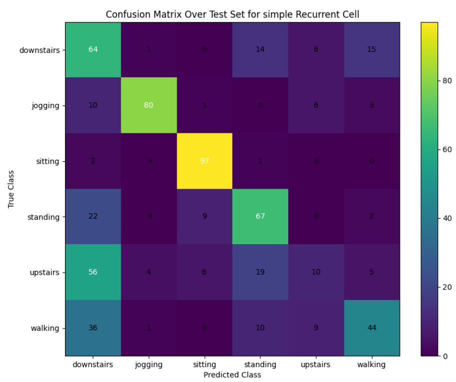
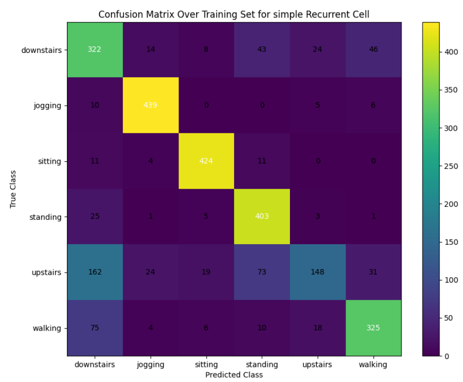


Figure 8: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.

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.

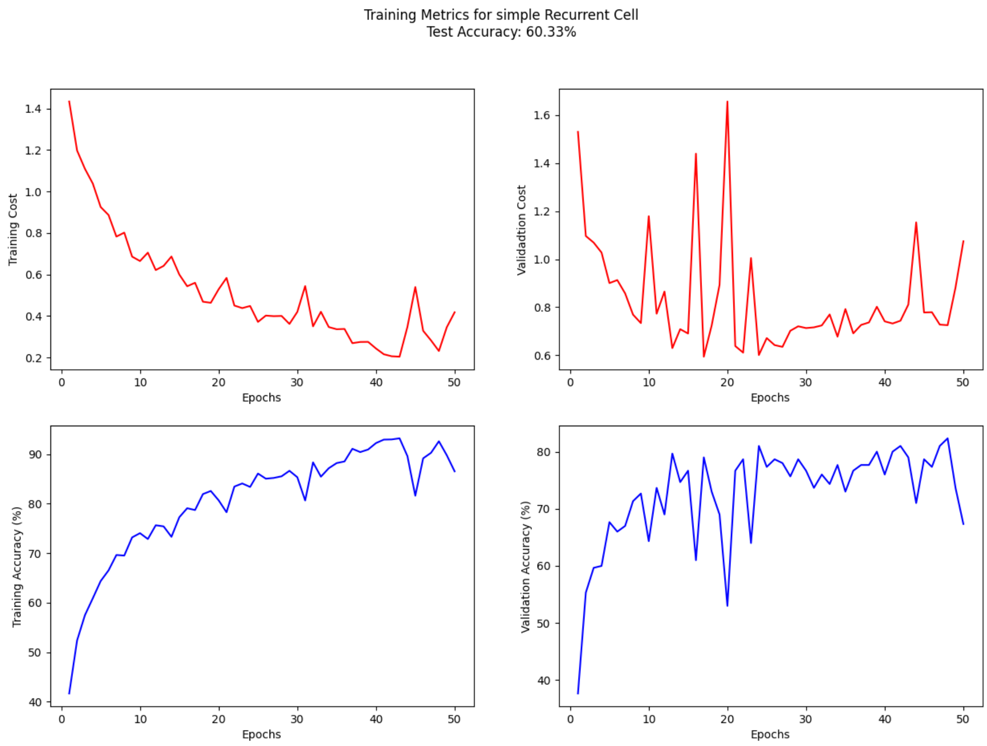


Figure 10: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number with BPTT truncation.

3.b) LSTM Implementation

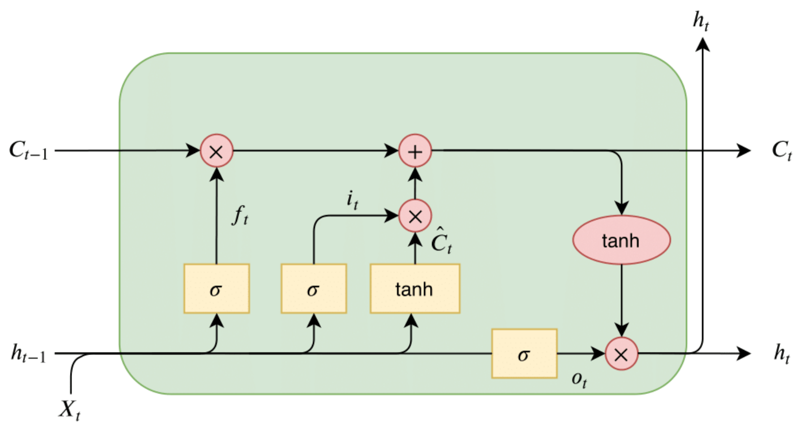


Figure 12: An illustration of the LSTM cell and its inner mechanisms.

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:

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

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.

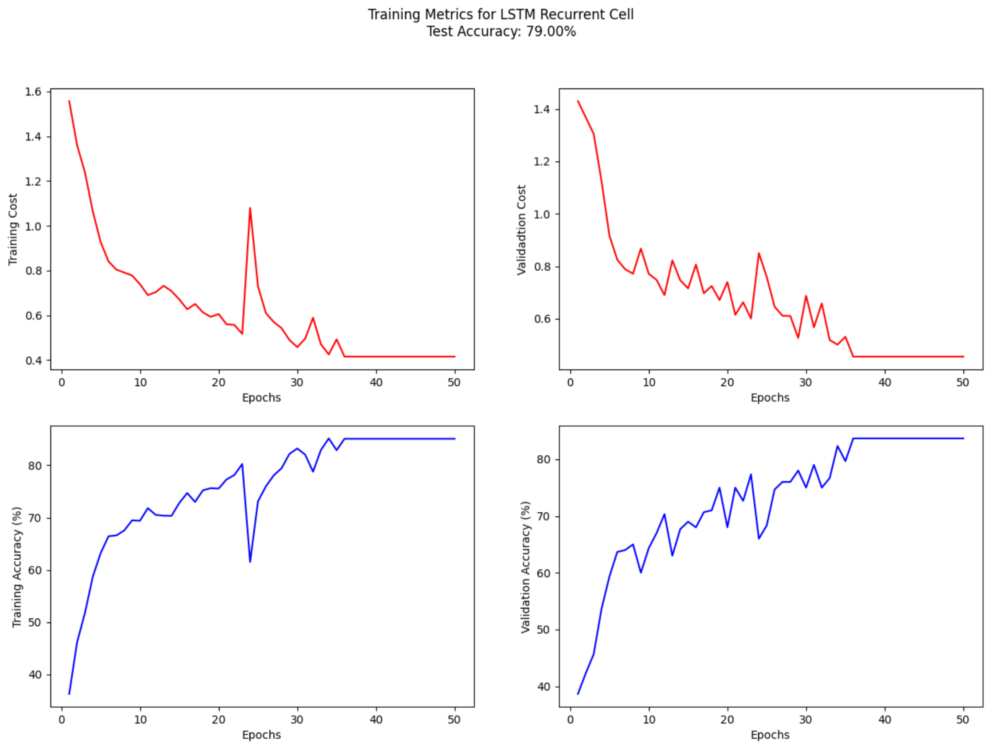
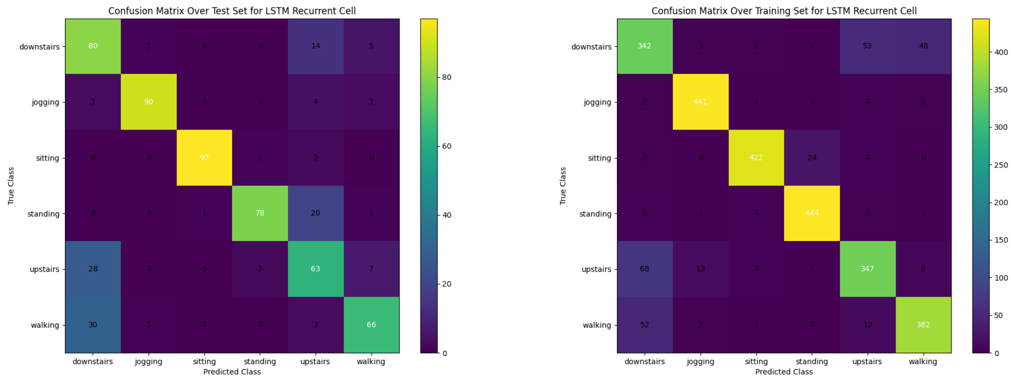


Figure 13: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.



(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

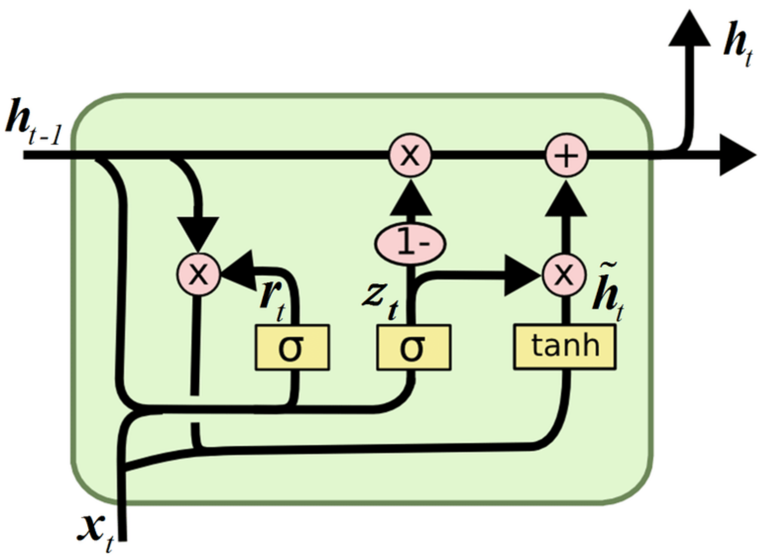


Figure 15: An illustration of the GRU cell and its inner mechanisms.

Gated recurrent unit, or GRU for short, was introduced as a simplification of LSTM cell with fewer parameters. It 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.

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

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.

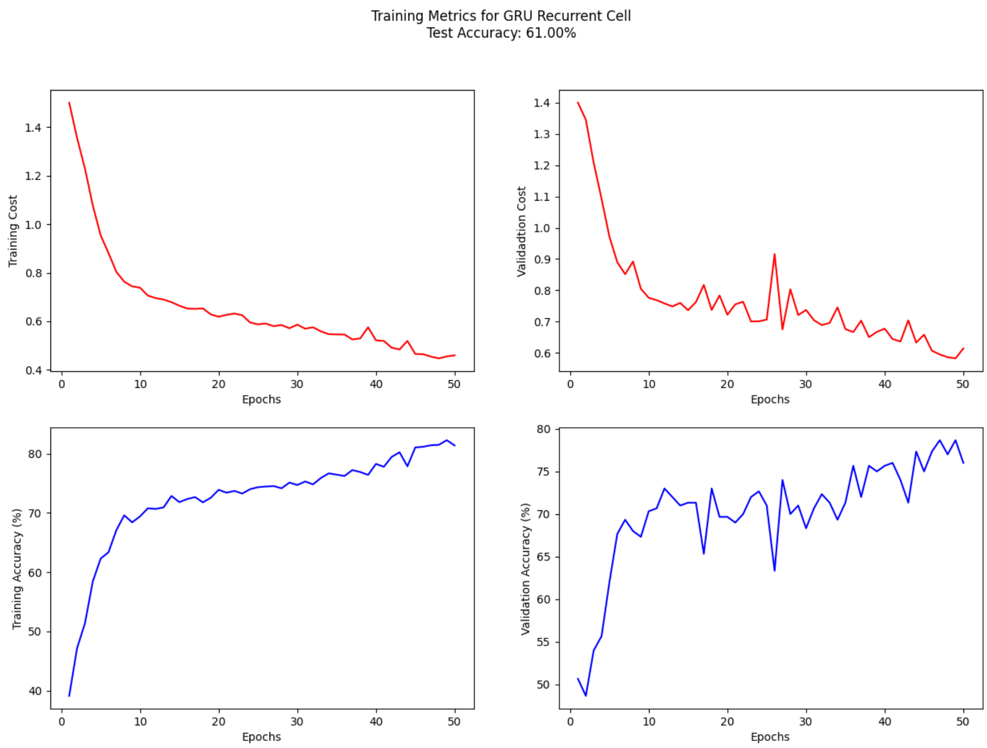
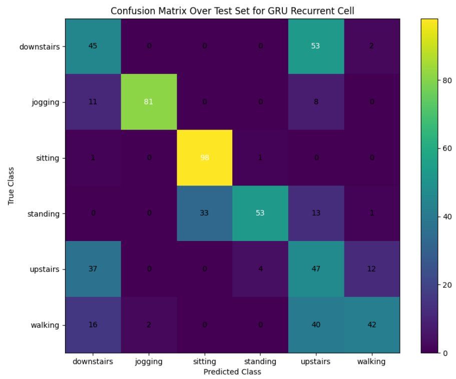
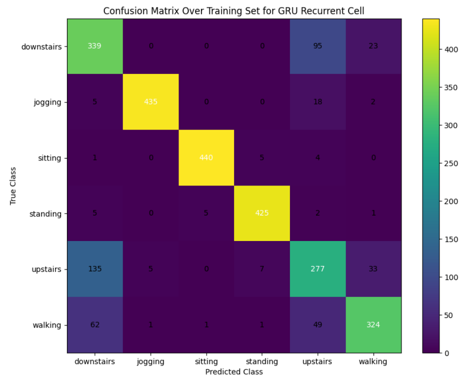


Figure 16: Cross-entropy cost and accuracy metrics for both training and validation data sets as a function of epoch number.



(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

```python
# Necessary imports
import numpy as np
import matplotlib.pyplot as plt
import h5py


#####################################################
################### CODE FOR Q-1 ####################
#####################################################

## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION

def sigmoid(X):
    """Returns the sigmoid function of the array X element-wise.

                sigmoid(X) = 1 / (1 + exp(-X))

    Args:
        X (np.array.ndarray): Input array.

    Returns:
        float: Sigmoid of X
    """
    return 1 / (1 + np.exp(-X))

def sigmoid_backward(X):
    """The derivative of the sigmoid function for back-propagation.

            sigmoid_backward(X) = exp(-X) / ((1 + exp(-X))^2)

    Args:
        X (np.arrray.ndarray): Input array

    Returns:
        _type_: The derivative of sigmoid at X
    """
    return np.exp(-X) / ((1 + np.exp(-X))**2)

def KL_div_ber(P, Q):
    """Returns the KL divergence between Bernoulli random variables with means P and Q.

                KL-div = (P * log(P / Q)) + ((1-P) * log((1-P) / (1-Q)))

    Args:
        P (float): Must be between 0 and 1.
```

```
        Q (float): Must be between 0 and 1.

    Returns:
        float: The KL divergence value.
    """
    return (P * np.log(P / Q)) + ((1-P) * np.log((1-P) / (1-Q)))

def grad_KL_div_ber(P, Q):
    """The derivative of the KL divergence of Bernoulli random variables with means P and Q, with respect to Q.

                            grad_KL_div_ber = (-P / Q) + ((1-P) / (1-Q))

    Args:
        P (float): Must be between 0 and 1.
        Q (float): Must be between 0 and 1.

    Returns:
        float: The derivative of KL divergence with respect to Q.
    """
    return (-P / Q) + ((1-P) / (1-Q))

def img_to_flat(data):
    """Takes an set of images, with sample dimention at last dimention, and flattens the images.

    Args:
        data (np.array.ndarray): Images with shape: (width, height, samples)

    Returns:
        np.array.ndarray: Flattened images with shape: (width * height, samples)
    """
    sample_size = data.shape[2]
    flat_size = data.shape[0] * data.shape[1]
    return np.reshape(data, (flat_size, sample_size))

def flat_to_img(data, img_shape):
    """Takes a set of data samples and reshapes the first dimention to the desired image shape

    Args:
        data (np.array.ndarray): Data with shape: (data, samples)
        img_shape (array): Desired image shape: (width, height)

    Raises:
        ValueError: It must hold that -> heights * width = data

    Returns:
```

```
        np.array.ndarray: Data reshaped to images of shape: (
            width, height, samples)
    """
    if data.shape[0] != img_shape[0] * img_shape[1]:
        raise ValueError('Need the shapes to match.')

    sample_size = data.shape[1]
    return np.reshape(data, (img_shape[0], img_shape[1],
       sample_size))

# Utility functions for creating the autoencoder network.
def init_AE_Wb(input_size, hidden_size):
    """Initializes the weights and biases for a single hidden
       layer autoencoder network. Uses Xavier initialization
       technique.

    Args:
        input_size (int): The input size, same as the output size
        hidden_size (int): The hidden layer size

    Returns:
        array: array that contains the weights and biasses: [W1,
            W2, b1, b2]
    """
    # Defining w0
    w0 = np.sqrt(6 / (input_size + hidden_size))

    # Random initialization of the parameters
    W1 = np.random.uniform(-w0, w0, (hidden_size, input_size))
    W2 = np.random.uniform(-w0, w0, (input_size, hidden_size))
    b1 = np.random.uniform(-w0, w0, (hidden_size, 1))
    b2 = np.random.uniform(-w0, w0, (input_size, 1))

    Wb = [W1, W2, b1, b2]

    return Wb

def aeCost(W_e, data, params):
    """Calculates the error of the autoencoder network with
       parameters W_e, for the given data.

    Args:
        W_e (array): Array containing the network parameters: [W1
            , W2, b1, b2]
        data (np.array.ndarray): The input data. Must be in shape
            : (L_in, samples)
        params (array): Array containing the additional
            information: [L_in, L_hid, cost_lambda, cost_beta,
            cost_rho]

    Returns:
```

```
        [float, dict]: array containg the net cost and a dictionary containing the gradients of
         cost with respect to network parameters and activations: [J, J_grad]
    """

    # Unpacking the network parameters
    W1, W2, b1, b2 = W_e

    batch_size = data.shape[1]

    L_in = params[0]
    L_hid = params[1]
    cost_lambda = params[2]
    cost_beta = params[3]
    cost_rho = params[4]

    # Forward propagation calculations
    A0 = data
    Z1 = np.matmul(W1, A0) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(W2, A1) + b2
    A2 = sigmoid(Z2)

    # Calculating the loss
    J_tto = (1 / (2 * batch_size)) * np.sum( (A2 - A0)**2 )
    J_Tykhonov = (cost_lambda / 2) * (np.sum( (W1)**2 ) + np.sum( (W2)**2 ))
    J_KL = cost_beta * np.sum(KL_div_ber(cost_rho, A1)) / batch_size
    J = J_tto + J_Tykhonov + J_KL

    # Calculating the derivatives
    dA2 = (A2 - A0) #/ batch_size
    dZ2 = dA2 * sigmoid_backward(Z2)
    dW2 = np.matmul(dZ2, A1.T) / batch_size + (cost_lambda * W2)
    db2 = np.sum(dZ2, axis=1, keepdims=True) / batch_size
    dA1 = np.matmul(W2.T, dZ2) + (cost_beta * grad_KL_div_ber(cost_rho, A1) / batch_size)
    dZ1 = dA1 * sigmoid_backward(Z1)
    dW1 = np.matmul(dZ1, A0.T) / batch_size + (cost_lambda * W1)
    db1 = np.sum(dZ1, axis=1, keepdims=True) / batch_size

    J_grad = {
        'dA2':dA2,
        'dZ2':dZ2,
        'dW2':dW2,
        'db2':db2,
        'dA1':dA1,
        'dZ1':dZ1,
        'dW1':dW1,
```

```python
        'db1':db1,
    }
    return [J, J_grad]

def solver(data, W_e, params, lr, batch_size, epochs, beta1=0.9,
   beta2=0.999, epsilon=1e-8, verbose=True):
    """Trains the network with parameters W_e with the given data
        using the Adam algorithm.

    Args:
        data (np.array.ndarray): Input data to be used for
           training. Shape must be (L_in, samples)
        W_e (array): Array with the initial network parameters: [
           W1, W2, b1, b2]
        params (array): Array containing the additional
           information: [L_in, L_hid, cost_lambda, cost_beta,
           cost_rho]
        lr (float): Learning rate for training. Advised to be
           below 1e-3
        batch_size (int): Number of samples used per parameter
           updating
        epochs (int): Number of iterations over the data
        beta1 (float, optional): Beta1 value for Adam optimizer.
           Defaults to 0.9.
        beta2 (float, optional): Beta2 value for Adam optimizer.
           Defaults to 0.999.
        epsilon (float, optional): Epsilon value for Adam
           optimizer. Defaults to 1e-8.
        verbose (bool, optional): For verbosing the training
           progress. Defaults to True.

    Returns:
        [[W1, W2, b1, b2], history]: The trained network
           parameters and the history of loss value for progress
           tracking
    """
    # Unpackinng the parameters
    W1, W2, b1, b2 = W_e

    # Initializing the momentum and velocity for Adam
    m_W1, m_W2, m_b1, m_b2 = 0, 0, 0, 0
    v_W1, v_W2, v_b1, v_b2 = 0, 0, 0, 0

    # Iteration  number for Adam
    t = 0

    sample_number = data.shape[1]
    passes_per_epoch = sample_number // batch_size

    # Initializing the loss history
    loss_history = np.zeros((epochs, 1))
```

```
    print('Starting training...\n')

    for i in range(epochs):

        epoch_loss = 0

        if verbose:
            print(f'Epoch: [{i+1}/{epochs}] -> ', end='')

        for j in range(passes_per_epoch):
            t += 1

            # Calculating the cost and the gradients for the batch
            J, J_grad = aeCost(W_e, data[:, j*batch_size:(j+1)*batch_size], params)
            epoch_loss += J

            # Updating the moemntum values
            m_W1 = beta1 * m_W1 + (1 - beta1) * J_grad['dW1']
            m_W2 = beta1 * m_W2 + (1 - beta1) * J_grad['dW2']
            m_b1 = beta1 * m_b1 + (1 - beta1) * J_grad['db1']
            m_b2 = beta1 * m_b2 + (1 - beta1) * J_grad['db2']

            # Updating the velocity values
            v_W1 = beta2 * v_W1 + (1 - beta2) * (J_grad['dW1'] ** 2)
            v_W2 = beta2 * v_W2 + (1 - beta2) * (J_grad['dW2'] ** 2)
            v_b1 = beta2 * v_b1 + (1 - beta2) * (J_grad['db1'] ** 2)
            v_b2 = beta2 * v_b2 + (1 - beta2) * (J_grad['db2'] ** 2)

            # Normalizing the momentum
            m_W1_hat = m_W1 / (1 - beta1 ** t)
            m_W2_hat = m_W2 / (1 - beta1 ** t)
            m_b1_hat = m_b1 / (1 - beta1 ** t)
            m_b2_hat = m_b2 / (1 - beta1 ** t)

            # Normalizing the velocity
            v_W1_hat = v_W1 / (1 - beta2 ** t)
            v_W2_hat = v_W2 / (1 - beta2 ** t)
            v_b1_hat = v_b1 / (1 - beta2 ** t)
            v_b2_hat = v_b2 / (1 - beta2 ** t)

            # Updating the weights and biases
            W1 -= lr * m_W1_hat / (np.sqrt(v_W1_hat) + epsilon)
            W2 -= lr * m_W2_hat / (np.sqrt(v_W2_hat) + epsilon)
            b1 -= lr * m_b1_hat / (np.sqrt(v_b1_hat) + epsilon)
```

```python
            b2 -= lr * m_b2_hat / (np.sqrt(v_b2_hat) + epsilon)

            # Updating W_e for the next batch of training
            W_e = [W1, W2, b1, b2]

        # Same training loop for the left-over samples that are left to the last batch
        if sample_number % batch_size != 0:
            t += 1

            J, J_grad = aeCost(W_e, data[:, (passes_per_epoch*batch_size):], params)
            epoch_loss += J

            m_W1 = beta1 * m_W1 + (1 - beta1) * J_grad['dW1']
            m_W2 = beta1 * m_W2 + (1 - beta1) * J_grad['dW2']
            m_b1 = beta1 * m_b1 + (1 - beta1) * J_grad['db1']
            m_b2 = beta1 * m_b2 + (1 - beta1) * J_grad['db2']

            v_W1 = beta2 * v_W1 + (1 - beta2) * (J_grad['dW1'] ** 2)
            v_W2 = beta2 * v_W2 + (1 - beta2) * (J_grad['dW2'] ** 2)
            v_b1 = beta2 * v_b1 + (1 - beta2) * (J_grad['db1'] ** 2)
            v_b2 = beta2 * v_b2 + (1 - beta2) * (J_grad['db2'] ** 2)

            m_W1_hat = m_W1 / (1 - beta1 ** t)
            m_W2_hat = m_W2 / (1 - beta1 ** t)
            m_b1_hat = m_b1 / (1 - beta1 ** t)
            m_b2_hat = m_b2 / (1 - beta1 ** t)

            v_W1_hat = v_W1 / (1 - beta2 ** t)
            v_W2_hat = v_W2 / (1 - beta2 ** t)
            v_b1_hat = v_b1 / (1 - beta2 ** t)
            v_b2_hat = v_b2 / (1 - beta2 ** t)

            W1 -= lr * m_W1_hat / (np.sqrt(v_W1_hat) + epsilon)
            W2 -= lr * m_W2_hat / (np.sqrt(v_W2_hat) + epsilon)
            b1 -= lr * m_b1_hat / (np.sqrt(v_b1_hat) + epsilon)
            b2 -= lr * m_b2_hat / (np.sqrt(v_b2_hat) + epsilon)

            W_e = [W1, W2, b1, b2]

        epoch_loss = (epoch_loss * batch_size) / sample_number
        loss_history[i] = epoch_loss

        if verbose:
                print(f'Epoch Loss: {epoch_loss:.4f}')

```

```python
    print('\nTraining completed.')

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

def predict(W_e, data):
    """Make predictions for the data with the network with
        parameters W_e

    Args:
        W_e (array): Array with the initial network parameters: [
            W1, W2, b1, b2]
        data (np.array.ndarray): Data in shape: (L_in, samples)

    Returns:
        np.array.ndarray: The network predictions of shape: (
            L_out, samples)
    """
    W1, W2, b1, b2 = W_e

    Z1 = np.matmul(W1, data) + b1
    A1 = sigmoid(Z1)

    Z2 = np.matmul(W2, A1) + b2
    A2 = sigmoid(Z2)

    return A2

# The function to perform the question-1 code
def q1():
    ### PART - 1.A
    ############################################
    # extracting the data from the data1.h5 file
    print('\nBegining Question-1 Part-A...\n')

    with h5py.File('data1.h5', 'r') as file:
        data_key = list(file.keys())[0]
        data = np.array(file[data_key])

    print('\nInitial Data Shape:', np.shape(data))

    # Carrying the channels to last
    data_channels_last = np.transpose(data, (0, 2, 3, 1))
    data_channels_last_scaled = (data_channels_last -
        data_channels_last.min()) / ((data_channels_last -
        data_channels_last.min()).max())

    print('Data With Channels-Last:', data_channels_last_scaled.
        shape)
    print('Scaled Data With Channels Max:',
        data_channels_last_scaled.max())
```

```
    print('Scaled Data With Channels Min:', data_channels_last_scaled.min())

    # scaling the data for the lumousity level
    data_gs = np.average(data, axis=1, weights=(0.2126, 0.7152, 0.0722))

    print('\nData Shape After Lumousity Scaling:', data_gs.shape)
    print('Gray-Scale Data Maximum Value:', data_gs.max())
    print('Gray-Scale Minumum Value:', data_gs.min())


    # Normalize the data by clipping the beyond three standart deviations
    data_gs_mean = np.mean(data_gs)
    data_gs_std = np.std(data_gs)

    print('\nData Average Value:', data_gs_mean)
    print('Data Standard Deviation:', data_gs_std)

    data_clip_limit_below = data_gs_mean - 3 * data_gs_std
    data_clip_limit_above = data_gs_mean + 3 * data_gs_std

    data_gs_normalized = np.clip(data_gs, data_clip_limit_below, data_clip_limit_above)

    print('\nClipped Data Shape:', data_gs_normalized.shape)
    print('Clipped Data Maximum Value:', data_gs_normalized.max())
    print('Clipped Data Minumum Value:', data_gs_normalized.min())


    # Scale the data to [0.1, 0.9] interval
    data_gs_norm_max = data_gs_normalized.max()
    data_gs_norm_min = data_gs_normalized.min()

    data_gs_scaled = (data_gs_normalized /((data_gs_norm_max - data_gs_norm_min) / 0.8)) \
    - ((data_gs_normalized /((data_gs_norm_max - data_gs_norm_min) / 0.8)).min() - 0.1)

    print('\nScaled Data Shape:', data_gs_scaled.shape)
    print('Scaled Maximum Value:', data_gs_scaled.max())
    print('Scaled Minumum Value:', data_gs_scaled.min())


    # Displaying the data
    sample_count = data_gs_scaled.shape[0]

    # Selecting the random samples
```

```
    random_samples = np.random.randint(sample_count, size=200)

    rows = 10
    columns = 20

    # Displaying the colored samples
    fig_gs = plt.figure(1, figsize=(16, 8))
    plt.title('Selected Gray-Scale Data Samples')

    for i in range(1, rows * columns + 1):
        fig_gs.add_subplot(rows, columns, i)
        plt.imshow(data_gs_scaled[random_samples[i-1]], cmap='
           gray')
        plt.axis('off')

    plt.subplots_adjust(hspace=0.01, wspace=0.01)
    plt.gca().set_axis_off()
    plt.savefig('Q1_A_gray_scale_images.png')

    # Displaying the gray-scale, normalized samples
    fig_colored = plt.figure(2, figsize=(16, 8))
    plt.title('Selected Colored Data Samples')

    for i in range(1, rows * columns + 1):
        fig_colored.add_subplot(rows, columns, i)
        plt.imshow(data_channels_last_scaled[random_samples[i
           -1]])
        plt.axis('off')

    plt.subplots_adjust(hspace=0.01, wspace=0.01)
    plt.gca().set_axis_off()
    plt.savefig('Q1_A_colored_images.png')

    print('-'*50)

    ## PART - 1.B
    #######################################
    # Shuffling the data samples
    print('\nBegining Question-1 Part-B...\n')

    np.random.shuffle(data_gs_scaled)
    print(f'\nData shape after shuffle: {data_gs_scaled.shape}')

    # Carrying the samples dimention to last for network
       dimentions
    data_gs_scaled = np.transpose(data_gs_scaled, (1, 2, 0))
    print(f'Data shape after reshaping: {data_gs_scaled.shape}')

    # Flattening the images for proper shape.
    model_input_train = img_to_flat(data_gs_scaled)

```

```
    # Network sizes
    input_size = 256
    hidden_size = 64

    # Initializing the weights and biases
    Wb = init_AE_Wb(input_size, hidden_size)

    # Assigning the cost parameters
    # Best values for cost_beta and cost_rho are 0.1 and 0.5
       respectively.
    cost_lambda = 5e-4
    cost_beta = 0.1
    cost_rho = 0.5
    params = [input_size, hidden_size, cost_lambda, cost_beta,
       cost_rho]

    # Batch size and epoch number
    batch_size = 32
    epochs = 50

    # Training the network for the training data
    Wb_trained, history = solver(model_input_train, Wb, params, 1
       e-3, batch_size, epochs)

    # x-axis for the loss history
    history_x_axis = np.arange(1, history.size+1)

    # Plotting the training losses
    fig_error = plt.figure(3, figsize=(5,5))
    plt.title('Training Error Values')
    plt.xlabel('Epochs')
    plt.ylabel('Error')
    plt.plot(history_x_axis, history)
    plt.savefig('Q1_B_loss_history.png')


    ### PART - 1.C
    ##################################################
    # Plotting the first layer connection weights
    print('\nBegining Question-1 Part-C...\n')

    W1 = Wb_trained[0].transpose()

    # Converting the connection weights from 1-D to 2-D array
    model_W1_img = flat_to_img(W1, (16, 16))
    model_W1_img = np.transpose(model_W1_img, (2, 0, 1))

    figure_latent = plt.figure(figsize=(12,12))
    plt.title(f'lambda = {cost_lambda}, L_hid = {hidden_size}')
    rows_latent = 8
    columns_latent = 8
```

```python

    # Plotting for every neuron in the hidden layer
    for i in range(1, rows_latent * columns_latent + 1):
        figure_latent.add_subplot(rows_latent, columns_latent, i)
        plt.imshow(model_W1_img[i-1], cmap='gray')
        plt.axis('off')

    plt.subplots_adjust(hspace=0.1, wspace=0.1)
    plt.gca().set_axis_off()
    plt.savefig('Q1_C_connection_weights.png')


    ### PART - 1.D
    ##################################################
    print('\nBegining Question-1 Part-D...\n')

    L_hid_values = [16, 49, 81]
    lambda_values = [1e-5, 1e-4, 1e-3]


    params = [input_size, hidden_size, cost_lambda, cost_beta, cost_rho]

    for lambda_value in lambda_values:
        for L_hid_value in L_hid_values:
            print(f'\nTraining for lambda={lambda_value} | L_hid={L_hid_value}\n')

            params = [input_size, L_hid_value, lambda_value, cost_beta, cost_rho]

            Wb = init_AE_Wb(input_size, L_hid_value)
            Wb_trained, history = solver(model_input_train, Wb, params, 1e-3, batch_size, epochs)

            W1 = Wb_trained[0].transpose()

            model_W1_img = flat_to_img(W1, (16, 16))
            model_W1_img = np.transpose(model_W1_img, (2, 0, 1))

            figure_latent = plt.figure(figsize=(12,12))
            plt.title(f'lambda={lambda_value} | L_hid = {L_hid_value}')
            rows_latent = int(np.sqrt(L_hid_value))
            columns_latent = int(np.sqrt(L_hid_value))

            for i in range(1, rows_latent * columns_latent + 1):
                figure_latent.add_subplot(rows_latent, columns_latent, i)
                plt.imshow(model_W1_img[i-1], cmap='gray')
                plt.axis('off')
```

```python

            plt.subplots_adjust(hspace=0.1, wspace=0.1)
            plt.gca().set_axis_off()
            plt.savefig(f'Q1_D_{lambda_value}_{L_hid_value}.png')


    plt.show()



#####################################################
################### CODE FOR Q-2 ####################
#####################################################

## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION

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

def sigmoid_backward(X):
    return np.exp(-X) / ((1 + np.exp(-X))**2)

def relu(X, alpha=0.01):
    return np.where(X>0, X, alpha * X)

def relu_backward(X, alpha=0.01):
    return np.where(X>0, 1, alpha)

def softmax(X, axis=0):
    max_vals = np.max(X, axis=axis, keepdims=True)
    e_X = np.exp(X - max_vals)
    Y = e_X / np.sum(e_X, axis=axis, keepdims=True)
    return Y


def init_NLP_Wb(dict_size, embed_size, hidden_size, std=0.01):
    """ Initializes the network parameters for the given embedding layer and hidden layer size.

    Args:
        dict_size (int): Number of words in the dictionary
        embed_size (int): Output dimnetion of the embedding layer
        hidden_size (int): Number of neurons in the hidden layer.
        std (float, optional): The standard deviation of of the Gaussian distribution from
        which the parameters are randomly sampled. Defaults to 0.01.

    Returns:
        dict: The dictionary that contains the initialized network parameters.
```

```python
    """

    W_embed = np.random.normal(0, std, (dict_size, embed_size))

    W1 = np.random.normal(0, std, (embed_size, hidden_size))
    W2 = np.random.normal(0, std, (hidden_size, dict_size))

    b1 = np.random.normal(0, std, (1, hidden_size))
    b2 = np.random.normal(0, std, (1, dict_size))

    params = {
        'W_embed': W_embed,
        'W1': W1,
        'W2': W2,
        'b1': b1,
        'b2': b2
    }

    return params

def NLP_forward_pass(params, data, dict_size):
    """ Performs forward propagation through the network.

    Args:
        params (dict): The dictionary of the network parameters.
        data (numpy.array.ndarray): The data to pass through the
           network.
        dict_size (int): Number of words in the dictionary

    Returns:
        dict: A cache of the results from the intermediate steps.
    """
    W_embed = params['W_embed']
    W1 = params['W1']
    W2 = params['W2']
    b1 = params['b1']
    b2 = params['b2']

    batch_size, context_size = data.shape
    data = np.eye(dict_size)[data]

    A_pre = np.sum(data, axis=1) / context_size

    A0 = np.matmul(A_pre, W_embed)

    Z1 = np.matmul(A0, W1) + b1
    A1 = sigmoid(Z1)

    Z2 = np.matmul(A1, W2) + b2
    A2 = softmax(Z2, axis=1)

```

```
    cache = {
        'A_pre': A_pre,
        'A0': A0,
        'Z1': Z1,
        'A1': A2,
        'Z2': Z2,
        'A2': A2
    }

    return cache


def nlpCost(params, data, labels, dict_size):
    """ Calculates the cost with respect to the data and the
       labels,
    and the gradient with respect to them.

    Args:
        params (dict): The dictionary of the network parameters.
        data (numpy.array.ndarray): The data to calculate cost
           against.
        labels (numpy.array.ndarray): The true labels for the
           data.
        dict_size (int): Number of words in the dictionary

    Returns:
        list: A list [J, grads] that contain average cost and
           gradients with respect to the parameters.
    """

    W_embed = params['W_embed']
    W1 = params['W1']
    W2 = params['W2']
    b1 = params['b1']
    b2 = params['b2']

    # convewrting the data and the labels to one-hot encodings.
    batch_size, context_size = data.shape
    data = np.eye(dict_size)[data]
    labels = np.eye(dict_size)[labels]

    # Forward propagation through the network
    A_pre = np.sum(data, axis=1)

    A0 = np.matmul(A_pre, W_embed)

    Z1 = np.matmul(A0, W1) + b1
    A1 = sigmoid(Z1)

    Z2 = np.matmul(A1, W2) + b2
    A2 = softmax(Z2, axis=1)
```

```
    # Cross-entropy cost calculation
    J = np.sum(-labels * np.log(A2)) / batch_size

    # Backpropagation
    dZ2 = (A2 - labels)
    dW2 = np.matmul(A1.T, dZ2) / batch_size
    db2 = np.sum(dZ2, axis=0, keepdims=True) / batch_size

    dA1 = np.matmul(dZ2, W2.T)
    dZ1 = sigmoid_backward(Z1) * dA1
    dW1 = np.matmul(A0.T, dZ1) / batch_size
    db1 = np.sum(dZ1, axis=0, keepdims=True) / batch_size

    dA0 = np.matmul(dZ1, W1.T) # = dZ0
    dW_embed = np.matmul(A_pre.T, dA0) / batch_size

    grads = {
        'dZ2': dZ2,
        'dW2': dW2,
        'db2': db2,
        'dA1': dA1,
        'dZ1': dZ1,
        'dW1': dW1,
        'db1': db1,
        'dA0': dA0,
        'dW_embed': dW_embed
    }
    return [J, grads]


def SGD_Q2(params, dict_size, data, labels, val_data, val_labels,
    lr, batch_size, stop_loss=0.1, max_epochs=50, alpha=0,
   verbose=True):
    """
    Implements stochastic gradient descent (SGD) for training a
       neural network to predict
    the fourth word in a sequence based on preceding trigrams.
       Here are some notes:
    - SGD_Q2 function implements weight updates using momentum.
    - Stops training if validation loss falls below `stop_loss`
       or `max_epochs` is reached.
    - Tracks and reports training and validation accuracy, as
       well as losses.
    - Assumes the use of auxiliary functions `nlpCost` and `
       NLP_forward_pass` for
      calculating cost, gradients, and predictions.

    Parameters
    ----------
    params : dict
```

```
        Dictionary containing model parameters (weights and
           biases).
    dict_size : int
        Size of the vocabulary.
    data : numpy.ndarray
        Training input data, where each row represents a trigram.
    labels : numpy.ndarray
        Training labels corresponding to the fourth word in each
           trigram.
    val_data : numpy.ndarray
        Validation input data for monitoring loss and accuracy.
    val_labels : numpy.ndarray
        Validation labels corresponding to val_data.
    lr : float
        Learning rate for the SGD algorithm.
    batch_size : int
        Size of mini-batches for SGD.
    stop_loss : float, optional
        Target validation loss for stopping criteria (default is
           0.1).
    max_epochs : int, optional
        Maximum number of training epochs (default is 50).
    alpha : float, optional
        Momentum factor to stabilize updates (default is 0).
    verbose : bool, optional
        If True, prints progress and metrics at each epoch (
           default is True).

    Returns
    -------
    list
        A list containing:
        - Updated model parameters after training (params).
        - Loss history during training (numpy.ndarray).
    """

    sample_number = data.shape[0]

    passes_per_epoch = sample_number // batch_size

    loss_history = np.ones((max_epochs * passes_per_epoch, 1)) *
       stop_loss

    print('Starting training...')
    print(f'Target Validation Loss: {stop_loss} | Maximum Number
       of Epochs: {max_epochs}.\n')

    epoch = 0
    val_loss = stop_loss + 1

    while val_loss > stop_loss and epoch < max_epochs:
```

```
        dW_embed, dW1, dW2, db1, db2 = 0, 0, 0, 0, 0

        perm = np.random.permutation(sample_number)
        data = data[perm, :]
        labels = labels[perm]

        epoch += 1
        epoch_loss = 0
        correct_labels = 0

        if verbose:
            print(f'Epoch: [{epoch}/{max_epochs}] -> ', end='')

        for batch in range(passes_per_epoch):

            batch_data = data[batch*batch_size:(batch+1)*batch_size, :]
            batch_labels = labels[batch*batch_size:(batch+1)*batch_size]

            J, J_grad = nlpCost(params, batch_data, batch_labels, dict_size)
            A = NLP_forward_pass(params, batch_data, dict_size)['A2']

            epoch_loss += J
            pred_labels = np.argmax(A, axis=1)
            correct_labels += np.where(pred_labels == batch_labels, 1, 0).sum()


            dW_embed = alpha * dW_embed - lr * J_grad['dW_embed']
            dW1 = alpha * dW1 - lr * J_grad['dW1']
            dW2 = alpha * dW2 - lr * J_grad['dW2']
            db1 = alpha * db1 - lr * J_grad['db1']
            db2 = alpha * db2 - lr * J_grad['db2']

            params['W_embed'] += dW_embed
            params['W1'] += dW1
            params['W2'] += dW2
            params['b1'] += db1
            params['b2'] += db2


            loss_history[(epoch-1)*passes_per_epoch + batch] = J


        if sample_number % batch_size != 0:

            batch_data = data[(passes_per_epoch*batch_size):, :]
```

```python
            batch_labels = labels[(passes_per_epoch*batch_size):]

            J, J_grad = nlpCost(params, batch_data, batch_labels, dict_size)
            A = NLP_forward_pass(params, batch_data, dict_size)['A2']

            epoch_loss += J
            pred_labels = np.argmax(A, axis=1)
            correct_labels += np.where(pred_labels == batch_labels, 1, 0).sum()

            dW_embed = alpha * dW_embed - lr * J_grad['dW_embed']
            dW1 = alpha * dW1 - lr * J_grad['dW1']
            dW2 = alpha * dW2 - lr * J_grad['dW2']
            db1 = alpha * db1 - lr * J_grad['db1']
            db2 = alpha * db2 - lr * J_grad['db2']

            params['W_embed'] += dW_embed
            params['W1'] += dW1
            params['W2'] += dW2
            params['b1'] += db1
            params['b2'] += db2

        val_loss, _ = nlpCost(params, val_data, val_labels, dict_size)
        val_A = NLP_forward_pass(params, val_data, dict_size)['A2']

        val_preds = np.argmax(val_A, axis=1)
        val_correct_labels = np.where(val_preds == val_labels, 1, 0).sum()
        val_total_labels = val_data.shape[0]

        train_acc_percent = (correct_labels / sample_number) * 100
        val_acc_percent = (val_correct_labels / val_total_labels) * 100
        epoch_loss = epoch_loss * batch_size / sample_number


        if verbose:
            print(f'Training Loss: {epoch_loss:.4f} | Training Accuracy: {train_acc_percent:.2f}% | Validation Loss: {val_loss:.4f} | Validation Accuracy: {val_acc_percent:.2f}%')


    print('\nTraining completed.')

    return [params, loss_history]
```

```


# The working code for the question-2
def q2():
    with h5py.File('data2.h5', 'r') as File:
        file_keys = list(File.keys())

        testd_ind = np.array(File[file_keys[0]]) - 1
        testx_ind = np.array(File[file_keys[1]]) - 1
        traind_ind = np.array(File[file_keys[2]]) - 1
        trainx_ind = np.array(File[file_keys[3]]) - 1
        vald_ind = np.array(File[file_keys[4]]) - 1
        valx_ind = np.array(File[file_keys[5]]) - 1
        words = np.array(File[file_keys[6]])

    num_words = words.shape[0]

    # Initializing the
    Wb_32_256 = init_NLP_Wb(num_words, 32, 256, 0.01)
    Wb_16_128 = init_NLP_Wb(num_words, 16, 128, 0.01)
    Wb_8_64 = init_NLP_Wb(num_words, 8, 64, 0.01)

    max_epochs = 50

    # Training for the instructed layer sizes
    print('\nTraining for -> Embedding Size = 32 | Hidden Size = 256...\n')
    Wb_32_256_trained, _ = SGD_Q2(Wb_32_256, num_words, trainx_ind, traind_ind, valx_ind, vald_ind, 0.15, 200, 2., alpha=0.85, max_epochs=max_epochs, verbose=True)

    print('\nTraining for -> Embedding Size = 16 | Hidden Size = 128...\n')
    Wb_16_128_trained, _ = SGD_Q2(Wb_16_128, num_words, trainx_ind, traind_ind, valx_ind, vald_ind, 0.15, 200, 2., alpha=0.85, max_epochs=max_epochs, verbose=True)

    print('\nTraining for -> Embedding Size = 8 | Hidden Size = 64...\n')
    Wb_8_64_trained, _ = SGD_Q2(Wb_8_64, num_words, trainx_ind, traind_ind, valx_ind, vald_ind, 0.15, 200, 2., alpha=0.85, max_epochs=max_epochs, verbose=True)


    # Making predictions on the test data
    pred_32_256 = NLP_forward_pass(Wb_32_256_trained, testx_ind, num_words)['A2']
    pred_16_128 = NLP_forward_pass(Wb_16_128_trained, testx_ind, num_words)['A2']
    pred_8_64 = NLP_forward_pass(Wb_8_64_trained, testx_ind, num_words)['A2']
```

```python

    # Retriving the 10 most expected words
    pred_32_256 = np.argsort(pred_32_256, axis=1)[:, -10:]
    pred_16_128 = np.argsort(pred_16_128, axis=1)[:, -10:]
    pred_8_64 = np.argsort(pred_8_64, axis=1)[:, -10:]

    # Sampling 5 random trigrams from the test data set
    test_size = testx_ind.shape[0]
    number_of_samples = 5
    random_samples = np.random.permutation(test_size)[:
        number_of_samples]

    for i in range(number_of_samples):
        print(f'\nTrigram: {words[testx_ind[random_samples[i]]]}
            |', f'True Value: {words[testd_ind[random_samples[i
            ]]]}')
        print(f'Top Predictions for Embedding Size = 32, Hidden
            Size = 256: ', [word for word in reversed(words[
            pred_32_256[i]])])
        print(f'Top Predictions for Embedding Size = 16, Hidden
            Size = 128: ', [word for word in reversed(words[
            pred_16_128[i]])])
        print(f'Top Predictions for Embedding Size = 8, Hidden
            Size = 64: ', [word for word in reversed(words[
            pred_8_64[i]])], '\n')


#####################################################
################### CODE FOR Q-2 #####################
#####################################################

## UTILITY FUNCTIONS THAT ARE USED IN THIS QUESTION


def softmax(X, axis=1):
    max_vals = np.max(X, axis=axis, keepdims=True)
    e_X = np.exp(X - max_vals)
    Y = e_X / np.sum(e_X, axis=axis, keepdims=True)
    return Y

def tanh_backward(X):
    return 1 - np.tanh(X)**2


def plot_confusion_matrix(predictions, true_labels, class_names=
    None):
    """
    Plots a confusion matrix for the given predictions and true
        labels without sklearn.

    Parameters:
```

```
    predictions (numpy.ndarray): Model predictions of shape (N,
        num_classes) (softmax outputs or similar).
    true_labels (numpy.ndarray): True labels of shape (N,
        num_classes) (one-hot encoded).
    class_names (array, optional): The names of the classes. If
        not provided, integers will be assigned.

    Returns:
    (matplotlib.figure.Figure): The confusion matrix plot.
    """
    # Convert one-hot encoded labels to integer labels
    num_classes = true_labels.shape[1]
    pred_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(true_labels, axis=1)

    # Initialize confusion matrix
    confusion_matrix = np.zeros((num_classes, num_classes), dtype
        =int)

    # Populate confusion matrix
    for t, p in zip(true_classes, pred_classes):
        confusion_matrix[t, p] += 1

    # Plot confusion matrix
    plot = plt.figure(figsize=(10, 8))
    plt.imshow(confusion_matrix, cmap='viridis')
    plt.colorbar()
    plt.xlabel("Predicted Class")
    plt.ylabel("True Class")

    if class_names == None:
        plt.xticks(ticks=np.arange(num_classes), labels=[f"Class
            {i+1}" for i in range(num_classes)])
        plt.yticks(ticks=np.arange(num_classes), labels=[f"Class
            {i+1}" for i in range(num_classes)])
    else:
        plt.xticks(ticks=np.arange(num_classes), labels=
            class_names)
        plt.yticks(ticks=np.arange(num_classes), labels=
            class_names)


    # Add numbers to each cell
    for i in range(num_classes):
        for j in range(num_classes):
            value = confusion_matrix[i, j]
            max_value = np.max(confusion_matrix) / 2

            if value > max_value:
                text_color = 'white'
            else:
```

```
                    text_color = 'black'

                plt.text(j, i, value, ha='center', va='center', color
                   =text_color)

    return plot




def initRecWb(input_size, hidden_size):
    """ Initializes the layer parameters for simple recurrent
    layer with Xavier initialization.

    Args:
        input_size (int): Number of input features.
        hidden_size (int): Number of hidden neurons.

    Returns:
        dict: The dictionary of the recurrent layer weights.
    """
    w0_ih = np.sqrt(6 / (input_size + hidden_size))
    W_ih = np.random.uniform(-w0_ih, w0_ih, (input_size,
       hidden_size))

    w0_hh = np.sqrt(6 / (2 * hidden_size))
    W_hh = np.random.uniform(-w0_hh, w0_hh, (hidden_size,
       hidden_size))

    b_ih = np.zeros((1, hidden_size))

    WbRec = {'W_hh': W_hh, 'W_ih': W_ih, 'b_ih': b_ih}
    return WbRec

def initSeqWb(sizes):
    """ Initializes the layer parameters for MLP
    layer with Xavier initialization.

    Args:
        sizes (array): the arrray of desired layer sizes.

    Returns:
        dict:  The dictionary of the layer weights for MLP.
    """
    Wb = {}

    for i in range(len(sizes) - 1):
            w_0 = np.sqrt(6 / (sizes[i] + sizes[i+1]))
            Wb['W' + str(i)] = np.random.uniform(-w_0, w_0, (
               sizes[i], sizes[i+1]))
            Wb['b' + str(i)] = np.zeros((1, sizes[i + 1]))
```

```python

    return Wb

def reccurentForward(WbRec, data):
    """
    Performs a forward pass through a recurrent neural network
        layer.

    Parameters
    ----------
    WbRec : dict
        Dictionary containing the recurrent network weights and
            biases:
        - 'W_ih': Input-to-hidden weight matrix.
        - 'W_hh': Hidden-to-hidden weight matrix.
        - 'b_ih': Bias vector for hidden states.
    data : numpy.ndarray
        Input data of shape (batch_size, time_steps, features),
            where:
        - batch_size: Number of samples in a batch.
        - time_steps: Number of time steps in the sequence.
        - features: Number of features at each time step.

    Returns
    -------
    list
        A list containing:
        - h_t: Hidden state of the last time step (numpy.ndarray)
            .
        - recCache: Dictionary containing intermediate
            calculations for backpropagation,
          including:
          - 'Zrec_<t>': Pre-activation values for each time step.
          - 'H_<t>': Hidden state for each time step.

    Notes
    -----
    - Uses the hyperbolic tangent (tanh) activation function for
        the hidden states.
    - Outputs the final hidden state and a cache for all time
        steps.
    """
    batch_size, time_steps, features = data.shape

    h_t = np.zeros((batch_size, WbRec['W_hh'].shape[0]))

    recCache = {}

    for t in range(time_steps):
        x_t = data[:, t, :]

```

```
        Z = np.matmul(x_t, WbRec['W_ih']) + np.matmul(h_t, WbRec['W_hh']) + WbRec['b_ih']
        h_t = np.tanh(Z)

        recCache['Zrec_' + str(t)] = Z
        recCache['H_' + str(t)] = h_t

    return [h_t, recCache]


def sequantialForward(Wb, data):
    """
    Performs a forward pass through a sequential multi-layer feedforward neural network.

    Parameters
    ----------
    Wb : dict
        Dictionary containing the weights and biases for each layer:
        - 'W<i>': Weight matrix for layer i.
        - 'b<i>': Bias vector for layer i.
    data : numpy.ndarray
        Input data of shape (batch_size, input_features), where:
        - batch_size: Number of samples in a batch.
        - input_features: Number of input features.

    Returns
    -------
    list
        A list containing:
        - A: Output of the final layer after the softmax activation (numpy.ndarray).
        - cache: Dictionary storing intermediate calculations for backpropagation, including:
          - 'A<i>': Activation output of layer i.
          - 'Z<i>': Pre-activation output of layer i.

    Notes
    -----
    - Applies the tanh activation function for all layers except the last.
    - The last layer uses a softmax activation for multi-class classification.
    - Outputs the final activation and a cache for backpropagation.
    """
    num_layers = len(Wb) // 2

    cache = {}

```

```python
    A = data

    cache['A-1'] = data
    cache['Z-1'] = np.zeros_like(data)

    for layer in range(num_layers - 1):
        Z = np.matmul(A, Wb['W' + str(layer)]) + Wb['b' + str(
            layer)]
        A = np.tanh(Z)

        cache['Z' + str(layer)] = Z
        cache['A' + str(layer)] = A

    Z = np.matmul(A, Wb['W' + str(num_layers - 1)]) + Wb['b' +
       str(num_layers - 1)]
    A = softmax(Z, axis=1)

    cache['Z' + str(num_layers - 1)] = Z
    cache['A' + str(num_layers - 1)] = A

    return [A, cache]


def sequentialBackward(Wb, labels, cache):
    """
    Performs backpropagation through a sequential multi-layer
       feedforward neural network.

    Parameters
    ----------
    Wb : dict
        Dictionary containing the weights and biases for each
           layer:
        - 'W<i>': Weight matrix for layer i.
        - 'b<i>': Bias vector for layer i.
    labels : numpy.ndarray
        One-hot encoded labels of shape (batch_size, num_classes)
           .
    cache : dict
        Dictionary containing forward pass intermediate
           calculations, including:
        - 'A<i>': Activation output of layer i.
        - 'Z<i>': Pre-activation output of layer i.

    Returns
    -------
    list
        A list containing:
        - J: Cross-entropy loss for the batch (float).
        - grads: Dictionary of gradients for weights and biases,
           including:
```

```
            - 'dW<i>': Gradient of weight matrix for layer i.
            - 'db<i>': Gradient of bias vector for layer i.
          - dA_prev (numpy.ndarray): Gradient of activation for inputing to an earlier layer's backpropagaiton.

    Notes
    -----
    - Computes the cross-entropy loss for multi-class classification.
    - Gradients are calculated for all weights and biases in the network.
    - Assumes the use of `tanh_backward` to compute the gradient of the tanh activation.
    """

    batch_size = labels.shape[0]

    num_layers = len(Wb) // 2
    # _, cache = sequantialForward(Wb, recLastState)

    lastActivation = cache['A' + str(num_layers-1)]

    J = -np.sum(labels * np.log(lastActivation)) / batch_size
    grads = {}

    A_prev = cache['A' + str(num_layers-1)]
    dZ = lastActivation - labels
    dA_prev = 0

    for layer in reversed(range(num_layers)):
        A_prev = cache['A' + str(layer-1)]
        grads['db' + str(layer)] = np.sum(dZ, axis=0, keepdims=True) / batch_size
        grads['dW' + str(layer)] = np.matmul(A_prev.T, dZ) / batch_size
        dA_prev = np.matmul(dZ, Wb['W' + str(layer)].T)
        dZ = dA_prev * tanh_backward(cache['Z' + str(layer-1)])

    return [J, grads, dA_prev]

def recBackward(WbRec, input, last_dA, recCache, max_lookback_distance=None):
    """
    Backpropagation through time for a single-layer RNN.

    Parameters
    ----------
    WbRec : dict
        Dictionary containing the recurrent layer parameters:
          - 'W_ih': Input-to-hidden weights, shape (D_in, D_h)
          - 'W_hh': Hidden-to-hidden weights, shape (D_h, D_h)
```

```
          - 'b_ih': Bias for hidden state, shape (D_h,)
    last_dA : np.ndarray
        The gradient of the loss w.r.t. the last hidden state, shape (N, D_h).
    recCache : dict
        Cache from the forward pass containing:
          - 'X_t', 'Zrec_t', 'H_t' for t=1,...,T
        Must contain all time steps that were used in forward propagation.
    max_lookback_distance : int or None
        The number of steps to backprop through time.
        If None or greater than the total number of steps, all steps are used.

    Returns
    -------
    grads : dict
        Dictionary containing:
          - 'dW_ih'
          - 'dW_hh'
          - 'db_ih'
    """

    batch_size = input.shape[0]

    W_ih = WbRec['W_ih']
    W_hh = WbRec['W_hh']
    b_ih = WbRec['b_ih']

    # Extract the max t by checking keys:
    timesteps = [int(k.split('_')[1]) for k in recCache.keys() if k.startswith('H_')]
    T = max(timesteps) if len(timesteps) > 0 else 0

    # If max_lookback_distance not provided or too large, use the full length
    if max_lookback_distance is None or max_lookback_distance > T:
        max_lookback_distance = T

    # Initialize gradients
    dW_ih = np.zeros_like(W_ih)
    dW_hh = np.zeros_like(W_hh)
    db_ih = np.zeros_like(b_ih)

    # The hidden state gradient at the final step
    dH_next = last_dA

    # Backprop through time
    for t in range(T, T - max_lookback_distance, -1):
        # Load cached values
```

```
        H_t = recCache[f'H_{t}']
        Z_t = recCache[f'Zrec_{t}']
        X_t = input[:, t, :]

        # For H_{t-1}, if t=1, we might have H_0 in cache or use
           zeros
        if t == 1:
            H_prev = recCache.get('H_0', np.zeros((X_t.shape[0],
               H_t.shape[1])))
        else:
            H_prev = recCache[f'H_{t-1}']

        # Compute dZ_t
        dZ_t = dH_next * tanh_backward(Z_t)

        # Compute gradients for parameters
        dW_ih += np.matmul(X_t.T, dZ_t) / batch_size
        dW_hh += np.matmul(H_prev.T, dZ_t) / batch_size
        db_ih += dZ_t.sum(axis=0) / batch_size  # (D_h,)

        dH_prev = np.matmul(dZ_t, W_hh.T)

        # Update dH_next for the next iteration
        dH_next = dH_prev

    recGrads = {
        'dW_ih': dW_ih,
        'dW_hh': dW_hh,
        'db_ih': db_ih
    }

    return recGrads

###########################
#### LSTM CODE ##########
#####################

def initLSTMWb(input_size, hidden_size):
    WbLSTM = {}
    w0_ih = np.sqrt(6 / (input_size + hidden_size))
    w0_hh = np.sqrt(6 / (2 * hidden_size))

    ih_names = ['W_f', 'W_i', 'W_o', 'W_c']
    hh_names = ['U_f', 'U_i', 'U_o', 'U_c']
    b_names = ['b_f', 'b_i', 'b_o', 'b_c']

    WbLSTM['W_f'] = np.random.uniform(-w0_ih, w0_ih, (input_size,
        hidden_size)),

    for name in ih_names:
```

```
        WbLSTM[name] = np.random.uniform(-w0_ih, w0_ih, (
            input_size, hidden_size))
    for name in hh_names:
        WbLSTM[name] = np.random.uniform(-w0_hh, w0_hh, (
            hidden_size, hidden_size))
    for name in b_names:
        WbLSTM[name] = np.zeros((1, hidden_size))

    return WbLSTM


def lstmForward(WbLSTM, data):
    batch_size, time_steps, features = data.shape

    W_f = WbLSTM['W_f']
    W_i = WbLSTM['W_i']
    W_o = WbLSTM['W_o']
    W_c = WbLSTM['W_c']

    U_f = WbLSTM['U_f']
    U_i = WbLSTM['U_i']
    U_o = WbLSTM['U_o']
    U_c = WbLSTM['U_c']

    b_f = WbLSTM['b_f']
    b_i = WbLSTM['b_i']
    b_o = WbLSTM['b_o']
    b_c = WbLSTM['b_c']


    # Initialize hidden state (h_t) and cell state (c_t) to zeros
    hidden_size = W_f.shape[1]
    h_t = np.zeros((batch_size, hidden_size))
    c_t = np.zeros((batch_size, hidden_size))

    lstmCache = {}

    for t in range(time_steps):
        x_t = data[:, t, :]

        # Compute gates
        f_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_f) + np.matmul(
            h_t, U_f) + b_f)))
        i_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_i) + np.matmul(
            h_t, U_i) + b_i)))
        o_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_o) + np.matmul(
            h_t, U_o) + b_o)))
        g_t = np.tanh(np.matmul(x_t, W_c) + np.matmul(h_t, U_c) +
             b_c)

        # Update cell state and hidden state
```

```
        c_t = f_t * c_t + i_t * g_t
        h_t = o_t * np.tanh(c_t)

        # Save intermediate values for backpropagation or debugging
        lstmCache['f_t_' + str(t)] = f_t
        lstmCache['i_t_' + str(t)] = i_t
        lstmCache['o_t_' + str(t)] = o_t
        lstmCache['g_t_' + str(t)] = g_t
        lstmCache['c_t_' + str(t)] = c_t
        lstmCache['h_t_' + str(t)] = h_t

    return [h_t, lstmCache]

def lstmBackward(WbLSTM, data, last_dA, lstmCache, max_lookback_distance=None):
    """
    Backpropagation through time for an LSTM layer.

    Parameters
    ----------
    WbLSTM : dict
        Dictionary containing the LSTM layer parameters:
          - 'W_f', 'W_i', 'W_o', 'W_c': Input-to-hidden weights, each shape (D_in, D_h)
          - 'U_f', 'U_i', 'U_o', 'U_c': Hidden-to-hidden weights, each shape (D_h, D_h)
          - 'b_f', 'b_i', 'b_o', 'b_c': Biases for gates, each shape (D_h,)
    data : np.ndarray
        Input data to the LSTM, shape (N, T, D_in)
    last_dA : np.ndarray
        Gradient of the loss w.r.t. the last hidden state, shape (N, D_h)
    lstmCache : dict
        Cache from the forward pass containing:
          - 'f_t_t', 'i_t_t', 'o_t_t', 'g_t_t', 'c_t_t', 'h_t_t' for t=1,...,T
        Must contain all time steps that were used in forward propagation.
    max_lookback_distance : int or None
        The number of steps to backprop through time.
        If None or greater than the total number of steps, all steps are used.

    Returns
    -------
    grads : dict
        Dictionary containing gradients for:
          - 'dW_f', 'dW_i', 'dW_o', 'dW_c'
          - 'dU_f', 'dU_i', 'dU_o', 'dU_c'
```

```
        - 'db_f', 'db_i', 'db_o', 'db_c'
    """
    N, T, D_in = data.shape
    hidden_size = WbLSTM['W_f'].shape[1]

    # Initialize gradients for weights, biases, and recurrent connections
    dW_f = np.zeros_like(WbLSTM['W_f'])
    dW_i = np.zeros_like(WbLSTM['W_i'])
    dW_o = np.zeros_like(WbLSTM['W_o'])
    dW_c = np.zeros_like(WbLSTM['W_c'])

    dU_f = np.zeros_like(WbLSTM['U_f'])
    dU_i = np.zeros_like(WbLSTM['U_i'])
    dU_o = np.zeros_like(WbLSTM['U_o'])
    dU_c = np.zeros_like(WbLSTM['U_c'])

    db_f = np.zeros_like(WbLSTM['b_f'])
    db_i = np.zeros_like(WbLSTM['b_i'])
    db_o = np.zeros_like(WbLSTM['b_o'])
    db_c = np.zeros_like(WbLSTM['b_c'])

    # Initialize gradients w.r.t. hidden and cell states
    dH_next = last_dA
    dC_next = np.zeros((N, hidden_size))

    # If max_lookback_distance is not provided, use the full sequence length
    if max_lookback_distance is None or max_lookback_distance > T:
        max_lookback_distance = T

    # Backprop through time
    for t in range(T - 1, T - max_lookback_distance - 1, -1):
        # Load cached values for time step t
        f_t = lstmCache[f'f_t_{t}']
        i_t = lstmCache[f'i_t_{t}']
        o_t = lstmCache[f'o_t_{t}']
        g_t = lstmCache[f'g_t_{t}']
        c_t = lstmCache[f'c_t_{t}']
        h_t = lstmCache[f'h_t_{t}']
        x_t = data[:, t, :]
        h_prev = lstmCache[f'h_t_{t-1}'] if t > 0 else np.zeros_like(h_t)
        c_prev = lstmCache[f'c_t_{t-1}'] if t > 0 else np.zeros_like(c_t)

        # Gradients w.r.t. cell state and output gate
        dO_t = dH_next * np.tanh(c_t)  # Gradient of output gate
        dC_t = dH_next * o_t * (1 - np.tanh(c_t)**2) + dC_next  # Gradient of cell state
```

```python

        # Gradients w.r.t. gates
        dF_t = dC_t * c_prev
        dI_t = dC_t * g_t
        dG_t = dC_t * i_t

        # Apply activation function derivatives
        dF_t *= f_t * (1 - f_t)  # Sigmoid derivative
        dI_t *= i_t * (1 - i_t)  # Sigmoid derivative
        dO_t *= o_t * (1 - o_t)  # Sigmoid derivative
        dG_t *= 1 - g_t**2       # Tanh derivative

        # Accumulate parameter gradients
        dW_f += np.matmul(x_t.T, dF_t) / N
        dW_i += np.matmul(x_t.T, dI_t) / N
        dW_o += np.matmul(x_t.T, dO_t) / N
        dW_c += np.matmul(x_t.T, dG_t) / N

        dU_f += np.matmul(h_prev.T, dF_t) / N
        dU_i += np.matmul(h_prev.T, dI_t) / N
        dU_o += np.matmul(h_prev.T, dO_t) / N
        dU_c += np.matmul(h_prev.T, dG_t) / N

        db_f += dF_t.sum(axis=0) / N
        db_i += dI_t.sum(axis=0) / N
        db_o += dO_t.sum(axis=0) / N
        db_c += dG_t.sum(axis=0) / N

        # Backpropagate into previous hidden and cell states
        dH_next = np.matmul(dF_t, WbLSTM['U_f'].T) + \
                  np.matmul(dI_t, WbLSTM['U_i'].T) + \
                  np.matmul(dO_t, WbLSTM['U_o'].T) + \
                  np.matmul(dG_t, WbLSTM['U_c'].T)

        dC_next = dC_t * f_t

    # Package gradients into a dictionary
    lstmGrads = {
        'dW_f': dW_f, 'dW_i': dW_i, 'dW_o': dW_o, 'dW_c': dW_c,
        'dU_f': dU_f, 'dU_i': dU_i, 'dU_o': dU_o, 'dU_c': dU_c,
        'db_f': db_f, 'db_i': db_i, 'db_o': db_o, 'db_c': db_c
    }

    return lstmGrads


########
# GRU CODE
##################

def initGRUWb(input_size, hidden_size):
```

```
    WbGRU = {}

    w0_hh = np.sqrt(6 / (2 * hidden_size))
    w0_ih = np.sqrt(6 / (input_size + hidden_size))

    WbGRU['W_z'] = np.random.uniform(-w0_ih, w0_ih, (input_size, hidden_size))
    WbGRU['W_r'] = np.random.uniform(-w0_ih, w0_ih, (input_size, hidden_size))
    WbGRU['W_h'] = np.random.uniform(-w0_ih, w0_ih, (input_size, hidden_size))

    WbGRU['U_z'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size, hidden_size))
    WbGRU['U_r'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size, hidden_size))
    WbGRU['U_h'] = np.random.uniform(-w0_hh, w0_hh, (hidden_size, hidden_size))

    WbGRU['b_z'] = np.zeros((1, hidden_size))
    WbGRU['b_r'] = np.zeros((1, hidden_size))
    WbGRU['b_h'] = np.zeros((1, hidden_size))

    return WbGRU

def gruForward(WbGRU, data):
    batch_size, time_steps, features = data.shape

    W_z = WbGRU['W_z']
    W_r = WbGRU['W_r']
    W_h = WbGRU['W_h']

    U_z = WbGRU['U_z']
    U_r = WbGRU['U_r']
    U_h = WbGRU['U_h']

    b_z = WbGRU['b_z']
    b_r = WbGRU['b_r']
    b_h = WbGRU['b_h']

    # Initialize hidden state (h_t) to zeros
    hidden_size = W_z.shape[1]
    h_t = np.zeros((batch_size, hidden_size))

    gruCache = {}

    for t in range(time_steps):
        x_t = data[:, t, :]

        # Compute gates
```

```python
        z_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_z) + np.matmul(h_t, U_z) + b_z)))
        r_t = 1 / (1 + np.exp(-(np.matmul(x_t, W_r) + np.matmul(h_t, U_r) + b_r)))
        h_hat_t = np.tanh(np.matmul(x_t, W_h) + np.matmul(r_t * h_t, U_h) + b_h)

        # Update hidden state
        h_t = z_t * h_t + (1 - z_t) * h_hat_t

        # Save intermediate values for backpropagation or debugging
        gruCache['z_t_' + str(t)] = z_t
        gruCache['r_t_' + str(t)] = r_t
        gruCache['h_hat_t_' + str(t)] = h_hat_t
        gruCache['h_t_' + str(t)] = h_t

    return [h_t, gruCache]


def gruBackward(WbGRU, data, last_dA, gruCache, max_lookback_distance=None):
    """
    Backpropagation through time for a GRU layer.

    Parameters
    ----------
    WbGRU : dict
        Dictionary containing the GRU layer parameters:
          - 'W_z', 'W_r', 'W_h': Input-to-hidden weights, each shape (D_in, D_h)
          - 'U_z', 'U_r', 'U_h': Hidden-to-hidden weights, each shape (D_h, D_h)
          - 'b_z', 'b_r', 'b_h': Biases for gates, each shape (D_h,)
    data : np.ndarray
        Input data to the GRU, shape (N, T, D_in)
    last_dA : np.ndarray
        Gradient of the loss w.r.t. the last hidden state, shape (N, D_h)
    gruCache : dict
        Cache from the forward pass containing:
          - 'z_t_t', 'r_t_t', 'h_hat_t_t', 'h_t_t' for t=1,...,T
        Must contain all time steps that were used in forward propagation.
    max_lookback_distance : int or None
        The number of steps to backprop through time.
        If None or greater than the total number of steps, all steps are used.

    Returns
```

```
    -------
    grads : dict
        Dictionary containing gradients for:
          - 'dW_z', 'dW_r', 'dW_h'
          - 'dU_z', 'dU_r', 'dU_h'
          - 'db_z', 'db_r', 'db_h'
    """
    N, T, D_in = data.shape
    hidden_size = WbGRU['W_z'].shape[1]

    # Initialize gradients for weights, biases, and recurrent connections
    dW_z = np.zeros_like(WbGRU['W_z'])
    dW_r = np.zeros_like(WbGRU['W_r'])
    dW_h = np.zeros_like(WbGRU['W_h'])

    dU_z = np.zeros_like(WbGRU['U_z'])
    dU_r = np.zeros_like(WbGRU['U_r'])
    dU_h = np.zeros_like(WbGRU['U_h'])

    db_z = np.zeros_like(WbGRU['b_z'])
    db_r = np.zeros_like(WbGRU['b_r'])
    db_h = np.zeros_like(WbGRU['b_h'])

    # Initialize gradients w.r.t. hidden state
    dH_next = last_dA

    # If max_lookback_distance is not provided, use the full sequence length
    if max_lookback_distance is None or max_lookback_distance > T:
        max_lookback_distance = T

    # Backprop through time
    for t in range(T - 1, T - max_lookback_distance - 1, -1):
        z_t = gruCache[f'z_t_{t}']
        r_t = gruCache[f'r_t_{t}']
        h_hat_t = gruCache[f'h_hat_t_{t}']
        h_t = gruCache[f'h_t_{t}']
        x_t = data[:, t, :]
        h_prev = gruCache[f'h_t_{t-1}'] if t > 0 else np.zeros_like(h_t)

        # Gradients w.r.t. hidden state
        dZ_t = dH_next * (h_prev - h_hat_t)
        dH_hat_t = dH_next * (1 - z_t)
        dH_prev = dH_next * z_t

        # Apply activation function derivatives
        dZ_t *= z_t * (1 - z_t)  # Sigmoid derivative
        dR_t = np.matmul(dH_hat_t, WbGRU['U_h'].T) * h_prev
```

```
            dR_t *= r_t * (1 - r_t)  # Sigmoid derivative
            dH_hat_t *= 1 - h_hat_t**2  # Tanh derivative

            # Accumulate parameter gradients
            dW_z += np.matmul(x_t.T, dZ_t) / N
            dW_r += np.matmul(x_t.T, dR_t) / N
            dW_h += np.matmul(x_t.T, dH_hat_t) / N

            dU_z += np.matmul(h_prev.T, dZ_t) / N
            dU_r += np.matmul(h_prev.T, dR_t) / N
            dU_h += np.matmul((r_t * h_prev).T, dH_hat_t) / N

            db_z += dZ_t.sum(axis=0) / N
            db_r += dR_t.sum(axis=0) / N
            db_h += dH_hat_t.sum(axis=0) / N

            # Backpropagate into previous hidden state
            dH_next = dH_prev + np.matmul(dZ_t, WbGRU['U_z'].T) + np.
               matmul(dR_t, WbGRU['U_r'].T)

    gruGrads = {
        'dW_z': dW_z, 'dW_r': dW_r, 'dW_h': dW_h,
        'dU_z': dU_z, 'dU_r': dU_r, 'dU_h': dU_h,
        'db_z': db_z, 'db_r': db_r, 'db_h': db_h
    }

    return gruGrads


###################
### SGD CODE AND UTILS
##############


def SGD_Q3(WbRec, Wb, data, labels, val_data, val_labels, lr,
   batch_size,
            stop_loss=0.1, max_epochs=50, alpha=0, verbose=True,
               recurrent_type='Simple', max_lookback_distance=None
               ):


    sample_number = data.shape[0]

    recFuncForward = reccurentForward
    recFuncBackward = recBackward

    # Determine the typr of recurrent layer.
    if recurrent_type == 'LSTM' or recurrent_type == 'lstm':
        recFuncForward = lstmForward
        recFuncBackward = lstmBackward
    elif recurrent_type == 'GRU' or recurrent_type == 'gru':
```

```
            recFuncForward = gruForward
            recFuncBackward = gruBackward

    # Initializa the dictionary that holds gradients.
    dParams = {}
    for param in Wb:
                dParams['d' + param] = np.zeros_like(Wb[param])
    for recParam in WbRec:
        dParams['d' + recParam] = np.zeros_like(WbRec[recParam])


    passes_per_epoch = sample_number // batch_size

    # Training metrics arrays
    train_loss_history = np.zeros((max_epochs, 1))
    val_loss_history = np.zeros((max_epochs, 1))
    train_accuracy_history = np.zeros((max_epochs, 1))
    val_accuracy_history = np.zeros((max_epochs, 1))

    J_train = 0
    J_val = 0
    val_acc_percent = 0
    train_acc_percent = 0

    print('Starting training...')
    print('Recurrency type:', recurrent_type)
    print(f'Target Validation Loss: {stop_loss} | Maximum Number of Epochs: {max_epochs}.\n')

    epoch = 0
    J_val = stop_loss + 1

    # Starting training
    while J_val > stop_loss and epoch != max_epochs:

        # Taking a random permutations of the data and the labels.
        perm = np.random.permutation(sample_number)
        data = data[perm, :]
        labels = labels[perm, :]

        J_train = 0
        J_val = 0

        correct_labels = 0
        total_labels = sample_number
        epoch += 1

        if verbose:
            print(f'Epoch: [{epoch}/{max_epochs}] -> ', end='')

```

```
        # Mini-batching
        for batch in range(passes_per_epoch):

            batch_data = batch*batch_size

            # Forward propagation through the network
            h_t, recCache = recFuncForward(WbRec, data[batch_data:batch_data + batch_size, :])
            A, cache = sequantialForward(Wb, h_t)

            # Cross-entropy cost calcualtion.
            J, grads, last_dA = sequentialBackward(Wb, labels[batch_data:batch_data + batch_size, :], cache)
            recGrads = recFuncBackward(WbRec, data[batch_data:batch_data + batch_size, :], last_dA, recCache, max_lookback_distance)

            # Calculating accuracy
            J_train += J
            pred_labels = np.argmax(A, axis=1)
            true_labels = np.argmax(labels[batch_data:batch_data + batch_size, :], axis=1)
            correct_labels += np.where(pred_labels == true_labels, 1, 0).sum()

            # Applying the momentum term
            for d_param in grads:
                dParams[d_param] = alpha * dParams[d_param] - lr * grads[d_param]
            for d_param in recGrads:
                dParams[d_param] = alpha * dParams[d_param] - lr * recGrads[d_param]

            # Updating the weights and biases
            for param in Wb:
                Wb[param] += dParams['d' + param]
            for param in WbRec:
                WbRec[param] += dParams['d' + param]


        # Doing the same steps for the residual batch.
        if sample_number % batch_size != 0:

            batch_data = passes_per_epoch*batch_size

            h_t, recCache = recFuncForward(WbRec, data[batch_data:, :])
            A, cache = sequantialForward(Wb, h_t)

            J, grads, last_dA = sequentialBackward(Wb, labels[batch_data:, :], cache)
```

```
            recGrads = recFuncBackward(WbRec, data[batch_data:,
               :], last_dA, recCache, max_lookback_distance)


            J_train += J
            pred_labels = np.argmax(A, axis=1)
            true_labels = np.argmax(labels[batch_data:batch_data
               + batch_size, :], axis=1)
            correct_labels += np.where(pred_labels == true_labels
               , 1, 0).sum()


            for d_param in grads:
                dParams[d_param] = alpha * dParams[d_param] - lr
                   * grads[d_param]
            for d_param in recGrads:
                dParams[d_param] = alpha * dParams[d_param] - lr
                   * recGrads[d_param]

            for param in Wb:
                Wb[param] += dParams['d' + param]
            for param in WbRec:
                WbRec[param] += dParams['d' + param]


        h_t, recCache = recFuncForward(WbRec, val_data)
        A, cache = sequantialForward(Wb, h_t)

        # Validation data set calculations
        J_val, _, _ = sequentialBackward(Wb, val_labels, cache)
        val_pred_labels = np.argmax(A, axis=1)
        val_true_labels = np.argmax(val_labels, axis=1)
        val_correct_labels = np.where(val_pred_labels ==
           val_true_labels, 1, 0).sum()
        val_total_labels = val_labels.shape[0]

        train_acc_percent = (correct_labels / total_labels) * 100
        val_acc_percent = (val_correct_labels / val_total_labels)
            * 100
        J_train = J_train * batch_size / sample_number

        # Updating the metrics arrays.
        train_loss_history[epoch-1] = J_train
        val_loss_history[epoch-1] = J_val
        train_accuracy_history[epoch-1] = train_acc_percent
        val_accuracy_history[epoch-1] = val_acc_percent

        if verbose:
            print(f'Training Cost: {J_train:.4f} | Training
               Accuracy: {train_acc_percent:.2f}% | Validation
               Cost: {J_val:.4f} | Validation Accuracy: {
```

```
                val_acc_percent:.2f}%')

    train_loss_history[epoch:] = J_train
    val_loss_history[epoch:] = J_val
    train_accuracy_history[epoch:] = train_acc_percent
    val_accuracy_history[epoch:] = val_acc_percent

    metrics = {'train_loss':train_loss_history, 'val_loss':val_loss_history,
                'train_accuracy':train_accuracy_history, 'val_accuracy':val_accuracy_history}

    print('\nTraining completed.')

    return [WbRec, Wb, metrics]


def get_accuracy(WbRec, Wb, testing_data, testing_labels, recurrency_type='Simple'):

    Forward = reccurentForward

    if recurrency_type == 'Simple' or recurrency_type == 'simple':
        Forward = reccurentForward
    elif recurrency_type == 'LSTM' or recurrency_type == 'lstm':
        Forward = lstmForward
    elif recurrency_type == 'GRU' or recurrency_type == 'gru':
        Forward = gruForward

    h_t, recCache = Forward(WbRec, testing_data)
    A, cache = sequantialForward(Wb, h_t)

    pred_labels = np.argmax(A, axis=1)
    true_labels = np.argmax(testing_labels, axis=1)

    correct = np.sum(np.where(true_labels == pred_labels, 1, 0))
    total = testing_data.shape[0]

    return [correct / total, A]


def initWbQ3(layer_sizes, recurrency_type):
    """ Initializes the network for the desired recurrency type and layer sizes

    Args:
        layer_sizes (_type_): _description_
        recurrency_type (array): The array of layer sizes.

    Returns:
```

```
        list: list containing the weights and biases:
            WbRec (dict): Recurrent layer parameters.
            Wb (dict): MLP layer parameters.
    """
    if recurrency_type == 'Simple' or recurrency_type == 'simple':
        initFuncWb = initRecWb
    elif recurrency_type == 'LSTM' or recurrency_type == 'lstm':
        initFuncWb = initLSTMWb
    elif recurrency_type == 'GRU' or recurrency_type == 'gru':
        initFuncWb = initGRUWb

    WbRec = initFuncWb(layer_sizes[0], layer_sizes[1])
    WbSeq = initSeqWb(layer_sizes[1:])

    return [WbRec, WbSeq]

def q3():
    """ The working code for question-3. Takes and returns no arguments.
    """
    with h5py.File('data3.h5', 'r') as file:
        trX = np.array(file['trX'])
        trY = np.array(file['trY'])
        tstX = np.array(file['tstX'])
        tstY = np.array(file['tstY'])


    # data preprecessing to normalize
    data_mean = trX.mean()
    data_std = trX.std()

    trX = (trX - data_mean) / data_std
    tstX = (tstX - data_mean) / data_std


    data_count = trX.shape[0]

    # train / validation splitting the data
    val_train_split = 0.1
    train_start_index = int(data_count * val_train_split)

    rand_perm = np.random.permutation(data_count)
    trX = trX[rand_perm]
    trY = trY[rand_perm]

    data_train = trX[train_start_index:]
    data_val = trX[:train_start_index]

    labels_train = trY[train_start_index:]
    labels_val = trY[:train_start_index]
```

```python

    print('Training data count:', data_train.shape[0])
    print('Validation data count:', data_val.shape[0], '\n')

    recurrent_types = ['Simple', 'LSTM', 'GRU']
    max_lookback_list = {'Simple':None, 'LSTM':None, 'GRU':None}
    layer_sizes = [3, 128, 64, 64, 6]
    class_names = ['downstairs', 'jogging', 'sitting', 'standing', 'upstairs', 'walking']

    # Model hyperparameters
    learning_rate = 0.01
    batch_size = 32
    max_epochs = 50
    stop_loss = 0.5
    alpha = 0.85
    epochs = np.arange(1, max_epochs+1, 1)


    for rec_type in recurrent_types:
        # initializing weights and biases
        WbRec, Wb = initWbQ3(layer_sizes, rec_type)

        # training the network
        trainedWbRec, trainedWb, loss_hist = SGD_Q3(WbRec, Wb,
                                                    data_train, labels_train,
                                                    data_val, labels_val,
                                                    lr=learning_rate, batch_size=batch_size, stop_loss=stop_loss,
                                                    max_epochs=max_epochs, alpha=alpha,
                                                    max_lookback_distance=max_lookback_list[rec_type],
                                                    recurrent_type=rec_type)

        # Displaying and saving the results
        print(f'\nCreating confusion matrix for {rec_type} recurrent cell test set...')
        test_accuracy, test_predictions = get_accuracy(trainedWbRec, trainedWb, tstX, tstY, rec_type)
        conf_mat = plot_confusion_matrix(test_predictions, tstY, class_names)
        plt.title(F'Confusion Matrix Over Test Set for {rec_type} Recurrent Cell')
        plt.savefig(f'conf_mat_{rec_type}_test.png')
```

```python

        print(f'\nCreating confusion matrix for {rec_type} recurrent cell training set...')
        train_accuracy, train_predictions = get_accuracy(trainedWbRec, trainedWb, data_train, labels_train, rec_type)
        conf_mat = plot_confusion_matrix(train_predictions, labels_train, class_names)
        plt.title(F'Confusion Matrix Over Training Set for {rec_type} Recurrent Cell')
        plt.savefig(f'conf_mat_{rec_type}_train.png')


        print(f'\nCreating the training costs and accuracy plot for {rec_type} recurrent cell...')
        metrics_figure = plt.figure(figsize=(15,10))

        metrics_figure.add_subplot(2,2,1)
        plt.plot(epochs, loss_hist['train_loss'], 'r')
        plt.xlabel('Epochs')
        plt.ylabel('Training Cost')

        metrics_figure.add_subplot(2,2,2)
        plt.plot(epochs, loss_hist['val_loss'], 'r')
        plt.xlabel('Epochs')
        plt.ylabel('Validadtion Cost')

        metrics_figure.add_subplot(2,2,3)
        plt.plot(epochs, loss_hist['train_accuracy'], 'b')
        plt.xlabel('Epochs')
        plt.ylabel('Training Accuracy (%)')

        metrics_figure.add_subplot(2,2,4)
        plt.plot(epochs, loss_hist['val_accuracy'], 'b')
        plt.xlabel('Epochs')
        plt.ylabel('Validation Accuracy (%)')

        plt.suptitle(f"Training Metrics for {rec_type} Recurrent Cell\nTest Accuracy: {100 * test_accuracy:.2f}%")
        plt.savefig(f'loss_and_acc_{rec_type}')

        print(f'\nTest accuracy for {rec_type} recurrent cell: {100 * test_accuracy:.2f}%\n')

    plt.show()



import sys

question = sys.argv[1]
```

```python

def ozan_cem_bas_22102757_mini_project(question):
    if question == '1':
        q1()
    elif question == '2':
        q2()
    elif question == '3':
        q3()
    else:
        print("Please enter '1', '2' or '3' in argv to run the corresponding questions program.")


ozan_cem_bas_22102757_mini_project(question)
```