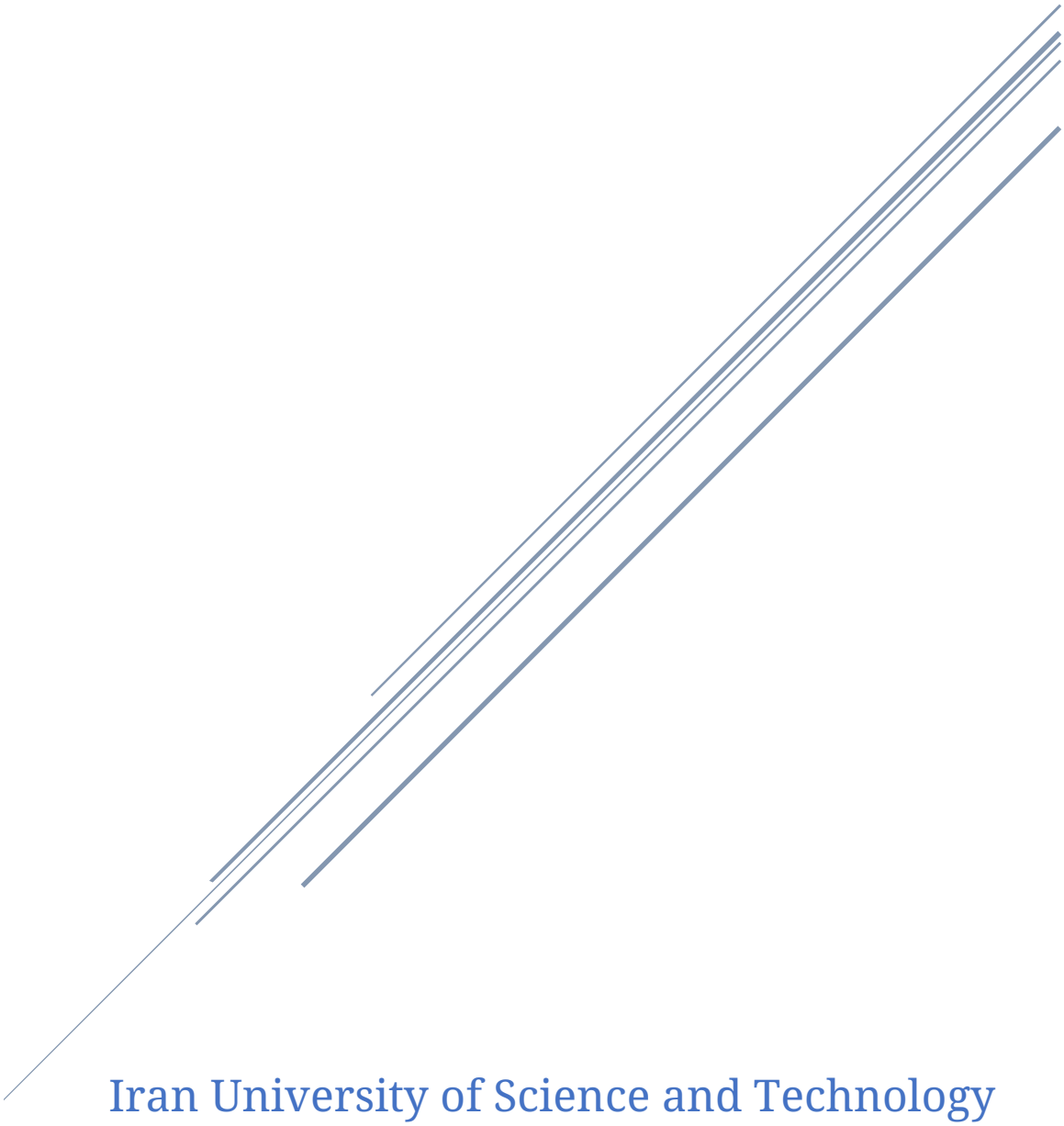
ANN 06 ASSIGNMENT

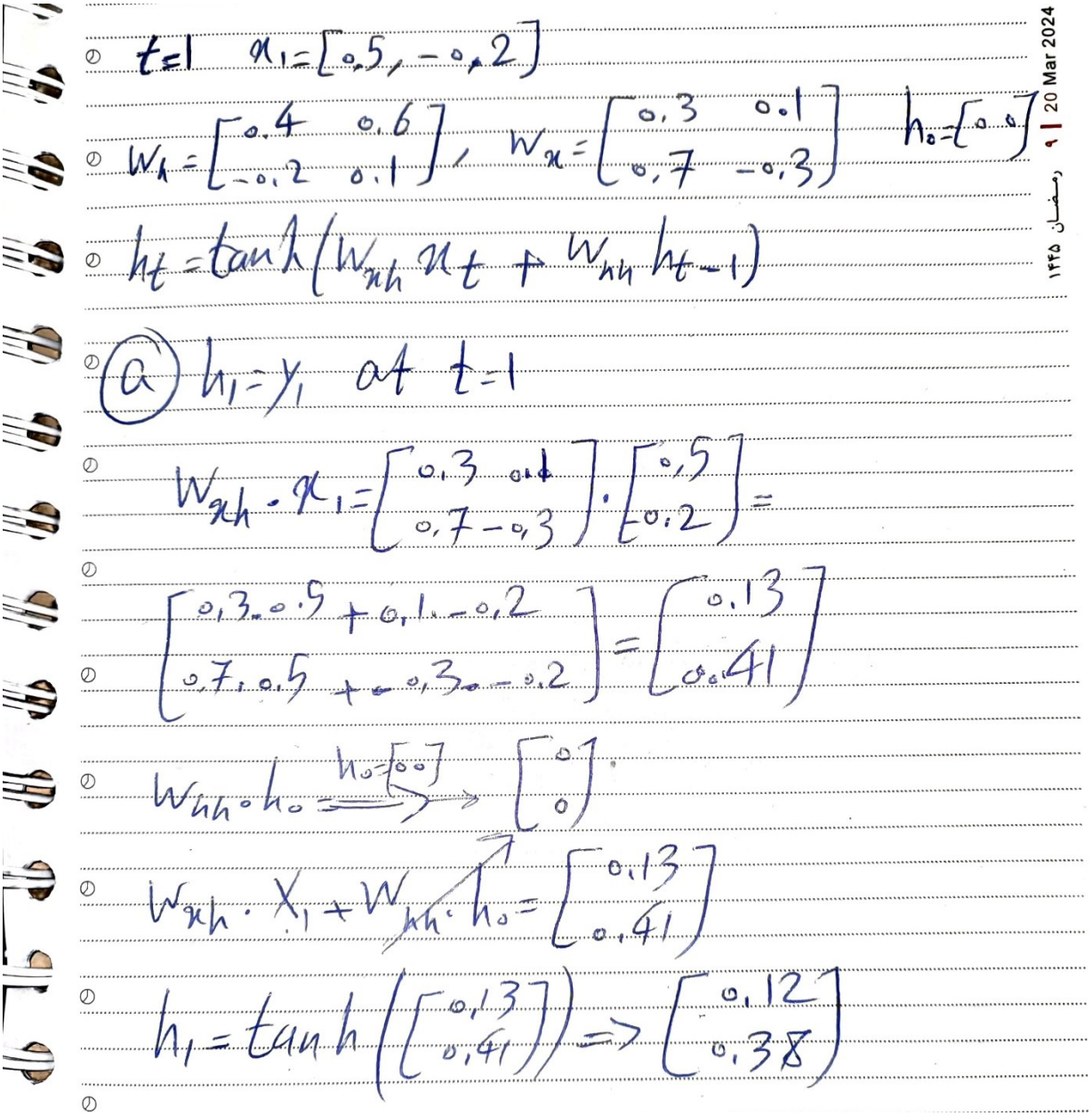
Hesam Koucheki | 403723987



Iran University of Science and Technology
Artificial Neural Network

Problem 1

Part A



Part B

The hidden state is responsible for keeping the important information of the previous steps and times while using less memory. In other words, it tries to map important informations in a vector that is representative of the previous data. While we go through further time steps, the network decides whether to keep features or update them in the feature map or the so-called vector. This hidden state allows the network to model temporal dependencies in the sequence of data while occupying less storage in comparison to keeping the original data. It also enables us faster computing since we do not need to process the previous data while we need them. In other say, in forward propagation, an RNN processes a sequence by iteratively updating its hidden state based on the current input and the previous hidden state. This mechanism enables it to capture patterns and dependencies in sequential data.

Steps for Forward Propagation

- ***Initialization***: The process starts with an initial hidden state h_0, often initialized as a zero vector or learned through training.
- ***Processing the Sequence:*** At each time step t, the RNN updates the hidden state using the formula below.

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

  The update involves:
  - **Compute the contribution from the current input x_t**; this captures how the current input affects the hidden state:

$$W_{xh} \cdot X_t$$

  - **Compute the contribution from the previous hidden state h_{t-1}**; this allows the RNN to remember information from prior time steps:

$$W_{hh} \cdot h_{t-1}$$

  - **Combine and apply activation**; The tanh activation ensures that the hidden state values remain within a manageable range (between -1 and 1) and introduces non-linearity:

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

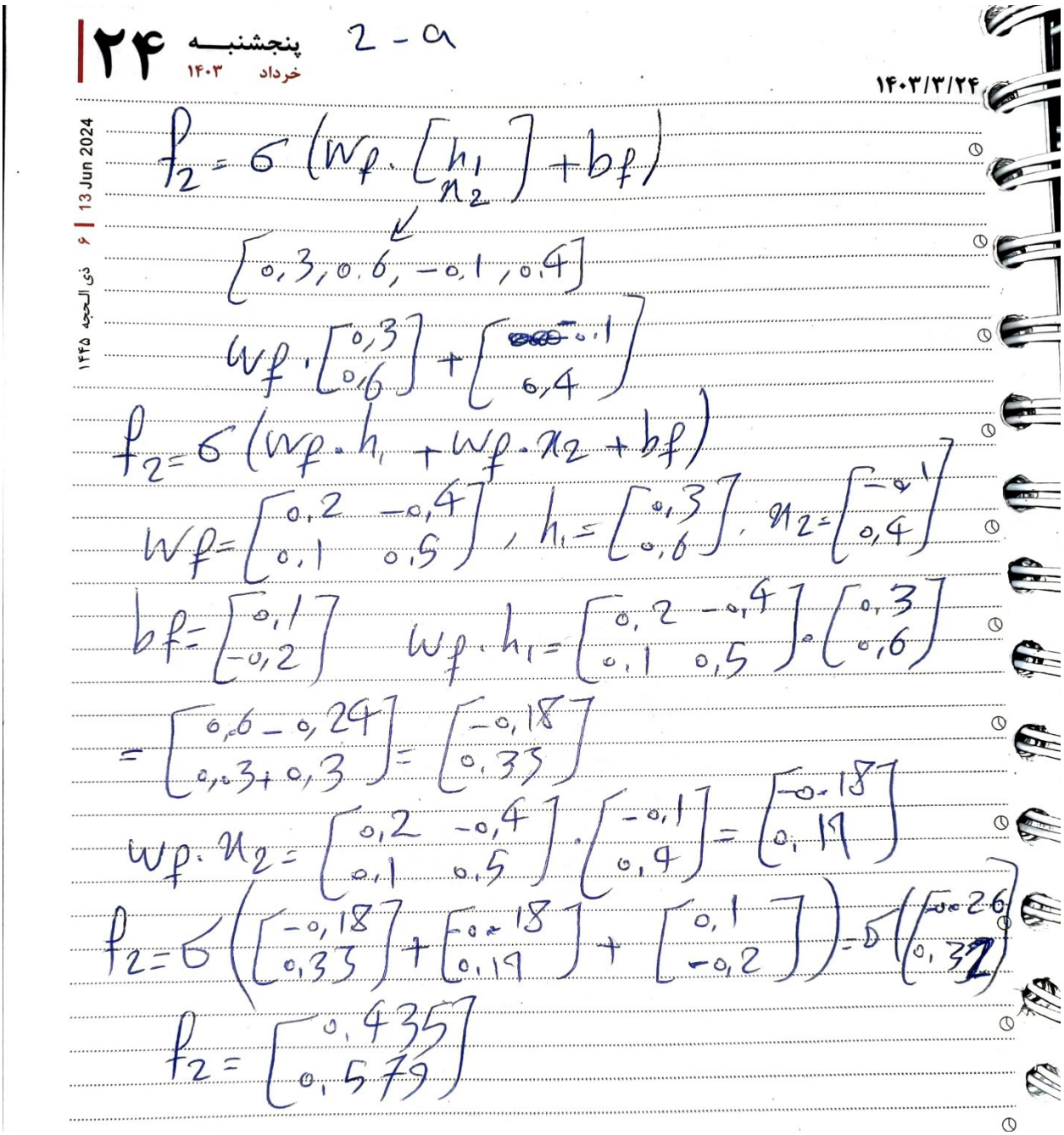
- ***Output***; At each time step, the RNN may produce an output y_t based on the hidden state:

$$y_t = W_{hy} \cdot h_t + b_y$$

- ***Repeat for all time steps*:** The above steps are repeated for each t in the sequence until the entire input sequence is processed.

Problem 2

Part A and B



$$i_2 = \sigma(W_{ij} \cdot h_1 + W_i \cdot x_2 + b_i)$$

$$W_i \cdot h_1 = \begin{bmatrix} -0.3 & 0.2 \\ -0.4 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.03 \\ 0.06 \end{bmatrix}$$

$$W_i \cdot x_2 = \begin{bmatrix} -0.3 & 0.2 \\ -0.4 & 0.3 \end{bmatrix} \begin{bmatrix} -0.1 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.11 \\ 0.16 \end{bmatrix}$$

$$W_i \cdot h_1 + W_i \cdot x_2 + b_i = \begin{bmatrix} 0.03 \\ 0.06 \end{bmatrix} + \begin{bmatrix} 0.11 \\ 0.16 \end{bmatrix} + \begin{bmatrix} -0.1 \\ 0.2 \end{bmatrix}$$

$$\rightarrow = \begin{bmatrix} 0.04 \\ 0.42 \end{bmatrix} \quad i = \sigma(x) \rightarrow i_2 = \begin{bmatrix} 0.51 \\ 0.6 \end{bmatrix}$$

$$g_2 = \tanh(W_c \cdot h_1 + W_c \cdot x_2 + b_c)$$

$$W_c \cdot h_1 = \begin{bmatrix} 0.6 & -0.1 \\ 0.5 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.12 \\ 0.33 \end{bmatrix}$$

$$W_c \cdot x_2 = \begin{bmatrix} 0.6 & -0.1 \\ 0.5 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} -0.1 \\ 0.4 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.07 \end{bmatrix}$$

$$W_c \cdot h_1 + W_c \cdot x_2 + b_c = \begin{bmatrix} 0.12 \\ 0.33 \end{bmatrix} + \begin{bmatrix} -0.1 \\ 0.07 \end{bmatrix} + \begin{bmatrix} 0.3 \\ -0.1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.32 \\ 0.3 \end{bmatrix} \quad g_2 = \tanh(x)$$

$$\Rightarrow g_2 = \begin{bmatrix} 0.309 \\ 0.291 \end{bmatrix}$$

$$C_2 = f_2 \odot C_1 + i_2 \odot g_2$$

$$f_2 \odot C_1 = \begin{bmatrix} 0.43 \\ 0.57 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ -0.3 \end{bmatrix} = \begin{bmatrix} 0.2175 \\ -0.1737 \end{bmatrix}$$

$$i_2 \odot g_2 = \begin{bmatrix} 0.51 \\ 0.6 \end{bmatrix} \odot \begin{bmatrix} 0.309 \\ 0.291 \end{bmatrix} = \begin{bmatrix} 0.157599 \\ 0.175473 \end{bmatrix}$$

$$C_2 \Rightarrow \begin{bmatrix} 0.37509 \\ 0.001773 \end{bmatrix}$$

2-b)

$$W_0 \cdot h_1 = \begin{bmatrix} -0.2 & 0.1 \\ 0.3 & -0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.21 \end{bmatrix}$$

$$W_0 \cdot h_2 = \begin{bmatrix} -0.2 & 0.1 \\ 0.3 & -0.5 \end{bmatrix} \cdot \begin{bmatrix} -0.1 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.06 \\ -0.23 \end{bmatrix}$$

$$W_0 \cdot h_1 + W_0 \cdot h_2 + b_0 = \begin{bmatrix} 0 \\ -0.21 \end{bmatrix} + \begin{bmatrix} 0.06 \\ -0.23 \end{bmatrix} + \begin{bmatrix} 0.05 \\ -0.05 \end{bmatrix}$$

$$= \begin{bmatrix} 0.11 \\ -0.49 \end{bmatrix} \xrightarrow{\text{Sigmoid}} O_2 = \begin{bmatrix} 0.5275 \\ 0.3799 \end{bmatrix}$$

$$h_2 = O_2 \odot \tanh(C_2)$$

$$\tanh(C_2) = \begin{bmatrix} 0.3584 \\ 0.001773 \end{bmatrix}$$

$$h_2 = \begin{bmatrix} 0.5275 \\ 0.3799 \end{bmatrix} \odot \begin{bmatrix} 0.3584 \\ 0.001773 \end{bmatrix} = \begin{bmatrix} 0.1891 \\ 0.000674 \end{bmatrix}$$

مادر پیاله عکس رخ یار دیده ایم — ای بی خبر ز لذت شرب مدام ما

Part C

RNNs were unable to store long-term dependencies, so LSTMs were invented to store long-term dependencies. The structure of an LSTM unit is cell state and three types of gates: the input gate, forget gate, and output gate, which will be described below:

Cell State

The cell state is equal to the memory of the LSTM, which holds and stores information for previous time steps. It has an information flow, enabling it to retain long-term dependencies effectively. The cell state is updated at each time step through the interaction of the gates. The cell state in an LSTM acts like a memory bank that helps the network remember important information over time. It carries information from one time step to the next, and the network can decide what to keep, what to forget, and what new information to add. The forget gate removes things it doesn't need, the input gate adds new important details, and this keeps the memory useful and focused. This is why LSTMs are great at understanding patterns that happen over a long period, like tracking events in a story or predicting trends in data.

Hidden State

As descibed before on previous questions. the hidden state represents the output of the LSTM unit at a particular time step. It is used to compute the output and is passed to the next time step along with the cell state to share the information and summary of the previous time steps. The hidden state in an LSTM is like a summary of what the network knows at a specific time step. It’s the output of the LSTM that combines the most important information from the current input and the past. Unlike the cell state, which acts like long-term memory, the hidden state is more like short-term memory, capturing details relevant to the current moment. This hidden state is passed to the next time step and can also be used to make predictions or outputs at each step. It helps the LSTM stay aware of both the immediate context and the bigger picture as it processes a sequence.

Input Gate

It determines which new information from the current input and previous hidden state should be stored in the cell state. The sigmoid activation (σ) outputs values between 0 and 1, where 0 means "block this information" and 1 means "allow this information." The input gate in an LSTM controls how much of the new information from the current input and the previous hidden state should be added to the cell state. It works like a filter (well, not the filter that we have on out Internet in IRAN 😊), deciding which parts of the incoming data are important enough to store in the memory. The input gate uses a sigmoid function to produce values between 0 and 1. This allows the LSTM to update its memory selectively, focusing on the relevant details while ignoring unnecessary noise, making it effective for processing sequences.

Forget Gate

It controls what information from the previous cell state (C_{t-1}) should be retained or discarded.
The forget gate decides, for each piece of information in the cell state, whether to "forget" it (closer to 0) or retain it (closer to 1). This enables the LSTM to selectively discard irrelevant information, ensuring that the memory is focused on important dependencies. The forget gate in an LSTM decides what information from the cell state should be removed or kept as the network processes new inputs. It works like a filter, using a sigmoid function to produce values between 0 and 1 for each piece of information in the cell state. A value closer to 0 means "forget this information," while

a value closer to 1 means "keep this information." This gate helps the LSTM focus on relevant details by clearing out unimportant or outdated information, ensuring the memory stays useful and efficient as the sequence progresses.

Output Gate

It Determines what part of the cell state should be output as the hidden state (ht). The output gate scales the activated cell state (tanh(Ct)) by the sigmoid output (o_t) to produce the hidden state.
This determines the information that the LSTM outputs to the next layer or time step. The output gate in an LSTM determines what information from the current cell state should be included in the hidden state, which serves as the output for the current time step. It uses a sigmoid function to decide the importance of different parts of the cell state, producing values between 0 and 1. Then, it multiplies these values with the (tanh) of the cell state to create the final hidden state. This gate ensures that the LSTM outputs relevant information based on both the current input and the long-term context stored in the cell state, making it effective for tasks like prediction and decision-making at each step of a sequence.

How do these gates help address the limitations of basic RNNs?

- ***Vanishing Gradient Problem:*** In basic RNNs, gradients diminish during backpropagation through time, especially for long sequences, making it hard to learn long-term dependencies. LSTMs solve this by using a nearly linear flow of information through the cell state. The forget and input gates regulate updates to the memory, ensuring gradients remain stable over long sequences.

- ***Long-Term Dependencies***: Basic RNNs have difficulty retaining important information over many time steps. LSTMs use the forget gate to remove irrelevant details from the memory and the input gate to selectively add new information, allowing the network to focus on significant patterns even far back in the sequence.

- ***Overwriting Information***: In basic RNNs, the hidden state is updated at every step, often leading to the loss of earlier information. LSTMs use the cell state as a dedicated memory mechanism, and the gates ensure that only useful updates are applied, preserving crucial information while discarding noise.

By dynamically controlling the flow of information, these gates enable LSTMs to effectively learn and process both short-term and long-term dependencies, making them perfect for sequential data tasks like time series prediction, language modeling.

Problem 3

First, we import various libraries that we may need. After that, we visualize and analyze the loaded dataset. By using df.head() we can see a few samples of the beginning of the dataset as well as their columns. Then by using df.isnull().sum() we can observe that how many of the featues are not. In the provided dataset, there is no null, so we do not need to perform any pre-processing to fill in the null features. After that, using df.duplicated().sum() shows us whether any data is duplicated or not, which we do not have any repetitive data. The code training_set = df.iloc[:,2:3].values extracts the target variable assumed to be the weather-related feature for prediction from the dataset as a NumPy array. This makes it suitable for further preprocessing and as input to the LSTM. By running the len(training_set) we can see that have 1461 samples for training our model.

df_to_XY

The function df_to_XY prepares input and output sequences for training an LSTM model by applying a sliding window approach. It takes a dataset and a window_size (default 10) as inputs. Inside the function, two empty lists, X_train and y_train, are initialized to store input sequences and corresponding target values. For each index starting from window_size to the end of the dataset, the function extracts a sequence of values of length window_size from the training_set as an input and appends it to X_train. It also extracts the value immediately following the sequence as the target and appends it to y_train. The lists are then converted into NumPy arrays for compatibility with models and returned as X_train and y_train. This function prepares sequential data for the LSTM by creating overlapping input-output pairs.

Regressor

This code snippet defines and compiles an LSTM-based regression model using Keras. It starts with a Sequential model, meaning layers are stacked in a linear sequence. The model begins with an LSTM layer containing 50 units, configured to return sequences (return_sequences=True) because additional LSTM layers follow. The input_shape is specified as (X_train.shape[1], 1), which indicates that each input sequence has a length equal to the number of time steps (X_train.shape[1]) and one feature per time step.

Three more LSTM layers with 50 units each are added, with the first two also returning sequences to feed into the next LSTM layer, while the final LSTM layer outputs a single vector. Between each LSTM layer, a Dropout layer is added with a dropout rate of 0.2, which helps reduce overfitting by randomly setting a fraction of the input units to zero during training.

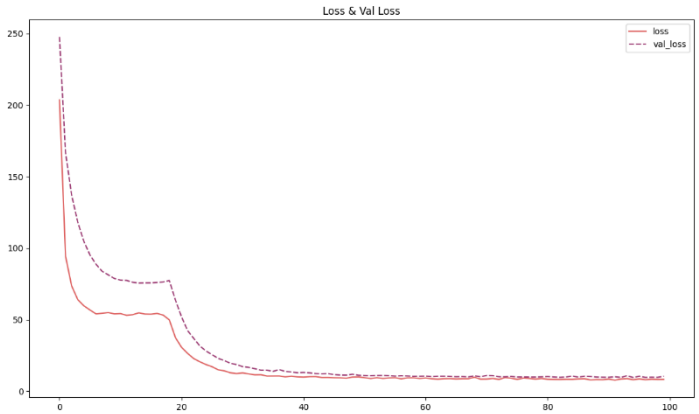
Finally, a Dense layer with one unit is added to produce a single output value, suitable for regression tasks. The model is compiled using the Adam optimizer, which adapts the learning rate during training for efficient convergence, and the loss function is set to mean squared error, appropriate for continuous output predictions. This architecture is designed to learn and predict time-series patterns effectively.

History

The code creates a Pandas DataFrame from the training history stored in history.history, which is a dictionary generated during model training with model.fit(). This dictionary contains the training metrics, such as the training loss (loss) and validation loss (val_loss), recorded at each epoch. By converting it into a DataFrame using pd.DataFrame(history.history), the information is organized in a tabular format where each column represents a metric and each row corresponds to an epoch. The his.head() command displays the first few rows of the DataFrame, allowing you to examine the initial values of the loss and validation loss during training. This transformation makes it easier to analyze the model's performance over the epochs and is often used for visualization or further analysis.

Loss & Val Loss

The graph represents the training and validation loss over 100 epochs during the training of the LSTM model. The red line represents the training loss, while the dashed purple line corresponds to the validation loss. Initially, both the training and validation loss start high, with a steep decline during the first 20 epochs, indicating that the model is learning effectively during the early stages of training.



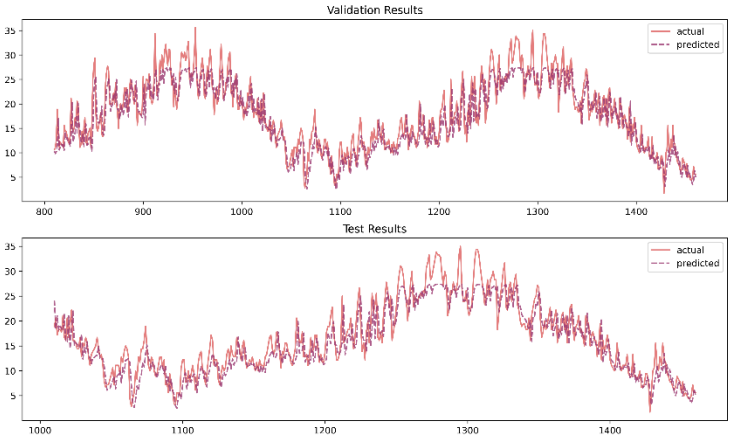
After epoch 20, the rate of decline slows down, and both the training and validation loss stabilize around similar values, indicating convergence. The training loss remains slightly lower than the validation loss, which is typical because the model is optimized specifically on the training data. Importantly, **there is no significant divergence between the two curves, which suggests that the model is not overfitting and has likely generalized well to the validation data.**

Overall, the plot indicates that the model training was successful, with both the training and validation loss decreasing steadily and converging to low values as the number of epochs increases.

Validation & Test Results

This image contains two plots showing the predicted and actual values for the LSTM weather prediction model on the validation and test datasets.

The upper graph, titled Validation Results, compares the actual and predicted values for the validation dataset after the LSTM model has been trained. The solid red line represents the actual values, while the dashed purple line corresponds to the model's predictions. This graph evaluates the model's performance on the validation set, which was not used for training but was used to monitor the model's progress during training.



The predictions in the validation graph align closely with the actual values, indicating that the model has successfully captured the underlying patterns in the data. The general trend of the predictions follows the actual data, including rises and falls, which suggests that the model understands the temporal dependencies in the weather dataset.

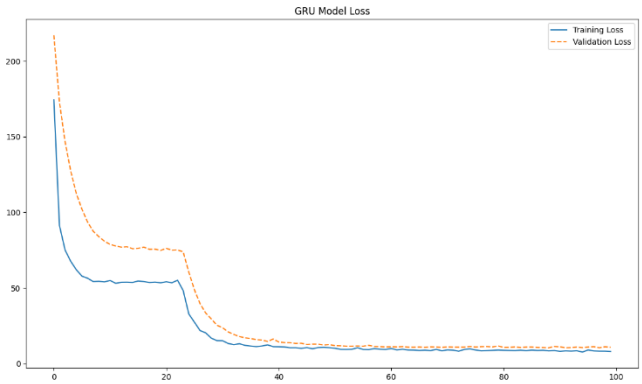
However, some minor deviations can be observed, especially during sharp peaks and troughs in the actual data. For instance, during periods of rapid changes or extremes, the model occasionally underestimates or overshoots the actual values. These inaccuracies are expected in real-world time-series predictions and could be attributed to the complexity of the patterns in the weather data or limitations in the model's capacity to capture abrupt changes.

Overall, the upper graph demonstrates that the model generalizes well to unseen validation data, providing reliable predictions for the majority of the time series, with slight inaccuracies at extreme points. This suggests the model's ability to learn the dataset's structure effectively while avoiding overfitting.

In the bottom plot, titled Test Results, the predicted values on the test dataset similarly follow the actual values closely. Although there are some deviations, particularly in sharp fluctuations and at extreme points, the predicted curve mirrors the overall trend of the actual values well. This indicates that the model is capable of making reasonable predictions even on unseen data.

GRU implementation and Loss

The implementation is available in the folder at the end of the Q3 file as well as their comments for the descriptions.
The graph shows the training and validation loss of the GRU-based model over 100 epochs. The blue solid line represents the training loss, while the orange dashed line represents the validation loss.

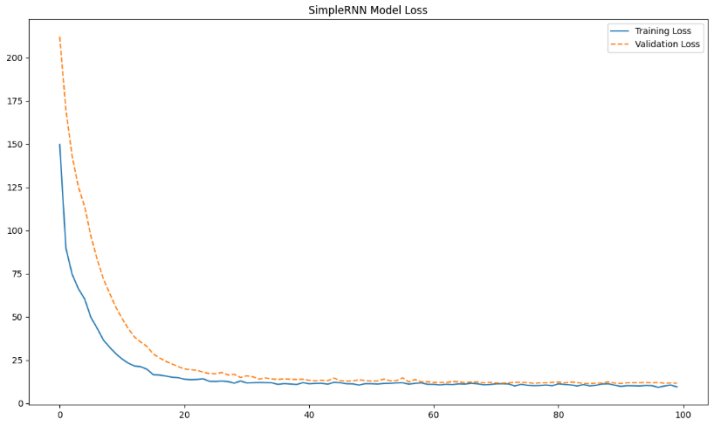


In the early epochs, both training and validation loss drop steeply, indicating that the model is learning the patterns in the data effectively. The validation loss follows the trend of the training loss, which suggests that the model is generalizing well and learning meaningful representations without significant overfitting.

Around epoch 20, both the training and validation loss start to stabilize, indicating that the model has reached a point where further learning does not yield significant improvement. The training loss remains slightly lower than the validation loss, which is typical because the model is optimized directly on the training data. However, the gap between training and validation loss is small, showing that the model is not overfitting and has good generalization ability.

SimpleRNN implementation and Loss

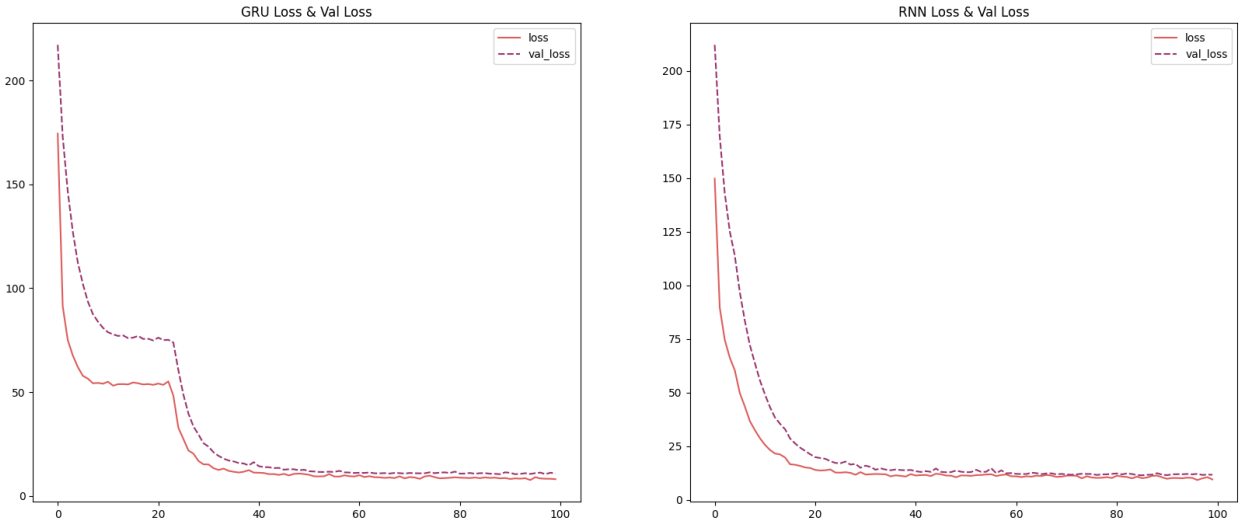
The implementation is available in the folder at the end of the Q3 file as well as their comments for the descriptions.
This graph shows the training and validation loss for the SimpleRNN model over 100 epochs. The blue solid line represents the training loss, while the orange dashed line corresponds to the validation loss.



In the initial epochs, both the training and validation loss drop steeply, indicating that the model is learning quickly. By around epoch 20, the rate of loss reduction slows, and the losses stabilize, suggesting that the model has reached convergence. The training loss consistently remains lower than the validation loss, which is expected because the model directly optimizes on the training data.

The gap between training and validation loss is minimal, which indicates that the model is not significantly overfitting the training data. This is a good sign for generalization, although the convergence behavior suggests that the model has limitations in capturing more complex dependencies.

Comparison



The two graphs compare the training loss and validation loss for the GRU and SimpleRNN models over 100 epochs.

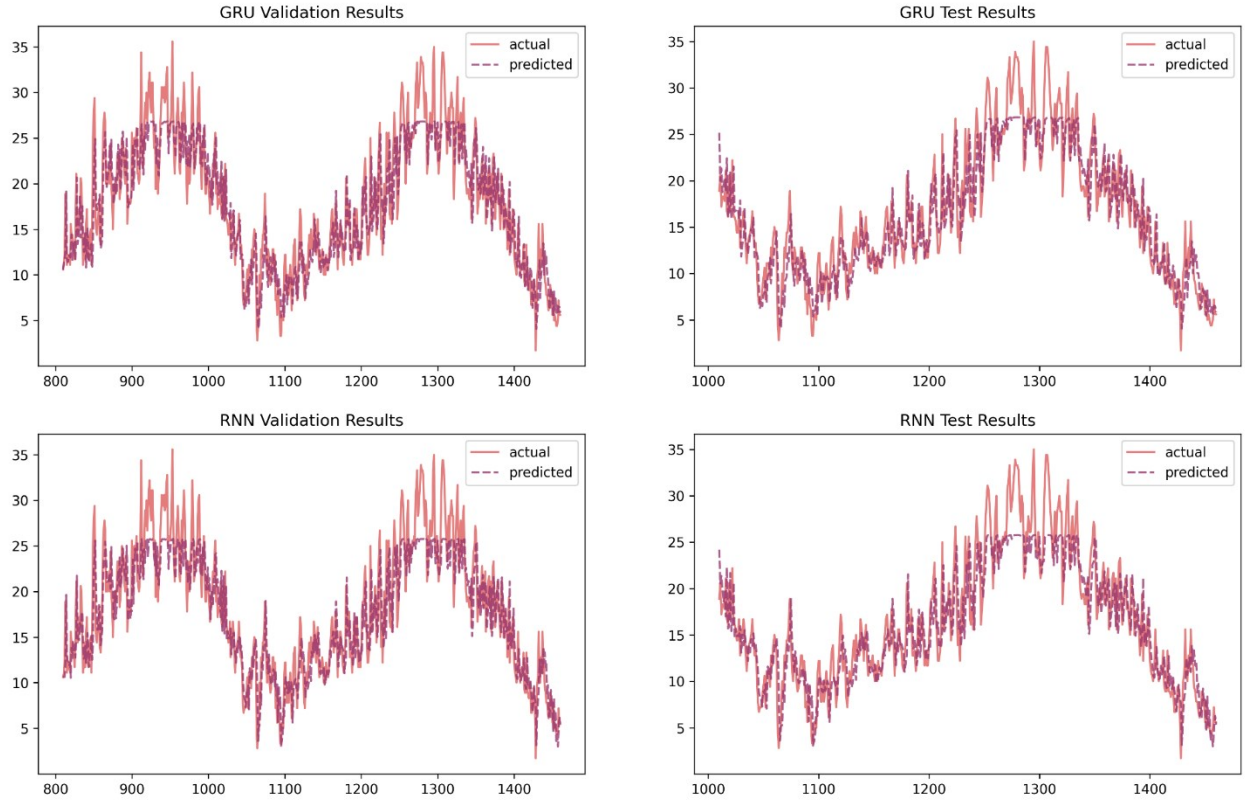
GRU Loss vs. RNN Loss

1. ***Convergence Speed:***
   - Both GRU and SimpleRNN models exhibit a steep drop in training and validation loss during the initial epochs, indicating rapid learning.
   - The GRU model converges faster and stabilizes more quickly than the SimpleRNN model, particularly after epoch 20. This suggests that the GRU's gating mechanisms allow it to learn temporal dependencies more efficiently.

2. ***Final Loss Values:***
   - The GRU model achieves lower training and validation loss compared to the SimpleRNN model, indicating better performance in terms of fitting the data and generalizing to the validation set. This can be attributed to the GRU's ability to handle long-term dependencies better.

3. ***Training vs. Validation Loss Gap:***
   - In both models, the gap between training loss and validation loss is small, showing that neither model is overfitting significantly. However, the GRU model demonstrates slightly better generalization, as its validation loss is closer to its training loss compared to SimpleRNN.

4. ***Stability:***
   - The GRU model's losses stabilize earlier (around epoch 30), while the SimpleRNN model's losses continue to decrease more gradually, reflecting slower convergence. This indicates that GRU is more robust and efficient in training.



This image contains four plots comparing the actual and predicted values for both the GRU and SimpleRNN models on the validation and test datasets.

GRU Validation Results (Top Left):

The GRU validation plot shows that the predicted values (dashed purple line) closely follow the actual values (solid red line). The GRU model effectively captures the overall trends and patterns in the validation data, including peaks and troughs. However, there are slight deviations during sharp fluctuations, particularly in extreme peaks, which are expected due to the inherent noise or variability in the data. Overall, the GRU model performs well on the validation dataset, demonstrating its ability to generalize the patterns it learned during training.

GRU Test Results (Top Right):

In the test dataset, the GRU model continues to perform well, with the predicted values closely matching the actual values. The model successfully captures the overall structure of the data, including rising and falling trends. The performance on the test set suggests that the GRU model generalizes well to unseen data, maintaining accuracy similar to its validation results.

SimpleRNN Validation Results (Bottom Left):

The SimpleRNN validation plot shows that the model can follow the general trend of the actual values but struggles with sharp peaks and abrupt changes. The deviations between the predicted

and actual values are more pronounced than those observed in the GRU model, especially in areas with high variability. This suggests that the SimpleRNN model has a limited capacity to learn complex temporal dependencies in the data, which aligns with its simpler architecture.

SimpleRNN Test Results (Bottom Right):

On the test dataset, the SimpleRNN model shows similar behavior to its validation performance. While it captures the general trends of the data, its predictions are less precise compared to the GRU model. The SimpleRNN model struggles with abrupt changes and extreme values, leading to larger deviations in certain regions. This indicates that the model has generalized moderately but lacks the robustness of the GRU model.

Overall Analysis:

The GRU model outperforms the SimpleRNN model on both the validation and test datasets. GRU provides more accurate predictions, effectively capturing trends and handling variability better than SimpleRNN. This is likely due to the gating mechanisms in GRU, which allow it to retain and manage long-term dependencies more efficiently. SimpleRNN, while able to capture basic patterns, shows limitations in modeling complex sequences, making it less suitable for this weather prediction task.

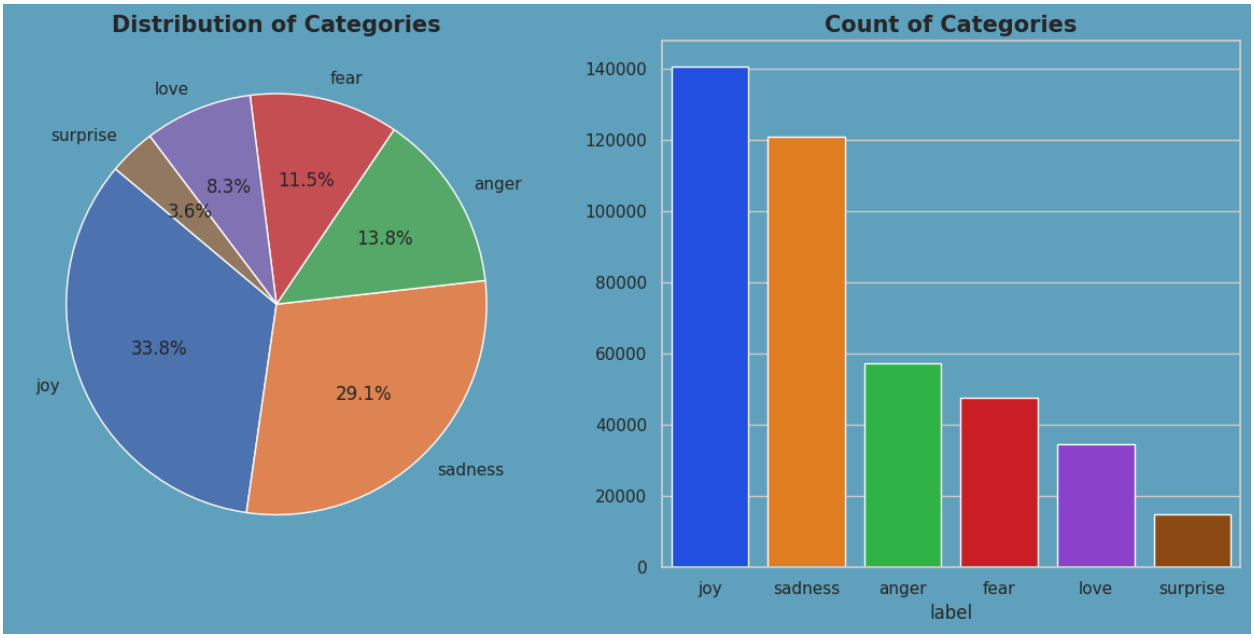
Problem 4

The code df.head(10) displays the first ten rows of the DataFrame df, which contains three columns: Unnamed: 0, text, and label. The Unnamed: 0 column is likely redundant and we further drop that column as it duplicates the index. The text column holds sentences expressing thoughts or emotions, such as "I just feel really helpless and heavy hearted," indicating the dataset is designed for an NLP task like sentiment or emotion classification. The label column contains numeric values representing categories or classes associated with the text, where each label likely corresponds to a specific emotion or sentiment. This dataset is suitable for supervised learning, with the text serving as input and label as the target for training a classification model. The Shape of data is (416809, 2) which means that we have about 416K samples of data with their corresponding label. By using df.duplicated().sum() we observe that there are 686 duplicated data which need to be removed to avoid any bias while training.

The provided code maps the numeric labels in the label column of the DataFrame df to their corresponding emotion names using the emotion_map dictionary. The mapping transforms the dataset to make it more interpretable and suitable for visualization.

The emotion_map dictionary associates each numeric label (e.g., 0, 1, 2) with its corresponding emotion category, such as sadness, joy, or fear. When df['label'].map(emotion_map) is executed, the numeric values in the label column are replaced with their corresponding emotion names. For example, a 0 in the label column would be replaced with sadness, and 1 would be replaced with joy.

After the mapping, the label column will contain descriptive emotion names instead of numeric codes. This transformation improves the readability of the data and makes it more intuitive to analyze and visualize, as visualizations like bar charts or pie charts will now display emotion categories instead of numeric labels.



The image provides a visualization of the distribution of emotion categories in the dataset using two charts: a pie chart and a bar chart.

Pie Chart: Distribution of Categories

The pie chart shows the proportional representation of each emotion category in the dataset. The largest segment, representing joy, constitutes 33.8% of the data, making it the most prevalent emotion. This is followed by sadness, which accounts for 29.1% of the data. Categories like anger (13.8%) and fear (11.5%) are less frequent but still substantial. The least represented categories are love (8.3%) and surprise (3.6%). The chart provides an intuitive way to understand the relative proportions of each emotion in the dataset.

Bar Chart: Count of Categories

The bar chart complements the pie chart by displaying the exact counts of samples for each emotion category. The height of each bar corresponds to the number of occurrences of a particular emotion. Consistent with the pie chart, joy and sadness have the highest counts, while love and surprise have the lowest. The chart makes it easier to compare the absolute frequencies between the categories.

Insights

The dataset appears to be imbalanced, with emotions like joy and sadness dominating the distribution. This imbalance may impact model training, as the model might perform better on the more frequent categories and struggle with underrepresented ones like surprise. Addressing this imbalance, for example, through data augmentation or sampling techniques, might improve the model's performance across all categories. Together, the pie chart and bar chart provide a clear understanding of the dataset's composition, aiding in the interpretation and preparation for further analysis.

Chat Words

A dictionary is called chat_words that maps common chat abbreviations or acronyms to their corresponding full forms. This dictionary is useful for preprocessing text data in tasks such as natural language processing or sentiment analysis, where abbreviations need to be expanded to improve the clarity and understanding of the text. For example the abbreviation "AFAIK" is mapped to "As Far As I Know". We further use the function replace_chat_words to replace chat words with their full forms in the texts for a more uniform training.

Data Cleaning

Removing Non-Alphanumeric Characters

The code provided focus on cleaning text data by removing non-alphanumeric characters. This step is essential in preparing text for analysis or modeling in natural language processing (NLP) tasks. The text column in the DataFrame (df['text']) is cleaned by applying a regular expression with the re.sub() function, which removes all characters that are not alphabets or spaces.

The code df['text'] = df['text'].apply(lambda x: re.sub(r'[^a-zA-Z\s]', '', x)) uses a regular expression pattern:
[^a-zA-Z\s]: Matches any character that is not a letter (a-z or A-Z) or a whitespace (\s).
The matched characters are replaced with an empty string (''), effectively removing them.
This process retains only the meaningful alphabetic content while discarding numbers, symbols, or special characters.

Removing stop words

We also replace the stop words with nothing “ ” to avoid the model focus on the redundant words that may have no effect on the training.

Lower case

We also lowercase all the letters to make it simpler and avoid the model distinguish between the lower and higher cases while they maintain the same meaning.

Removing numerics

After that the numerics are removed since they have few affects on our tasks which is emotion processing.

Removing white spaces

We also remove the whitespaces to make it cleaner.

Removing the special characters

Since we are working with emotions, we may not need the special characters.

Removing urls

Web addresses are completely removed from the data to avoid existence of any dirty data.

Tokenizing

Tokenization is a foundational step in natural language processing (NLP) where text is broken down into smaller units called tokens, such as words or subwords. This process is handled by a Tokenizer, which converts raw text into a structured format suitable for machine learning models. By splitting text into tokens, tokenization makes it easier to preprocess and analyze textual data for tasks like text classification, sentiment analysis, and machine translation. Additionally, tokenization ensures standardization, enabling machine learning algorithms to handle textual inputs efficiently.

In NLP, a sequence refers to an ordered list of tokens that represents a sentence or a piece of text. Maintaining the order of tokens in sequences is crucial because the meaning of a sentence often depends on the arrangement of its words. For instance, word order affects context and relationships, which are critical for models to understand and process language effectively. Sequences allow NLP models to learn from the inherent sequential structure of text, which is essential for tasks like language modeling, text generation, and machine translation.

The importance of sequences lies in their ability to capture the context and relationships between words, enabling models to make sense of the underlying meaning. By leveraging the sequential nature of language, models can improve performance in tasks requiring a deeper understanding

of text, such as predicting the next word in a sentence or analyzing sentiment. Tokenization and sequences thus play a pivotal role in enabling machines to process, learn from, and generate human language.
The Tokenizer is initialized with num_words=60000, meaning it will only keep the top 60,000 most frequent words from the dataset while ignoring less common ones. This helps reduce the vocabulary size, making the model more efficient and preventing it from overfitting on rare words. The texts_to_sequences method converts each text sample into a list of integers, where each integer represents a word's token. For example, a sentence like "I love programming" might become [2, 45, 105] depending on the word index.
The maxlen variable is used to store the maximum sequence length in X_train_sequences, which is essential for padding or truncating sequences later. To calculate it:

```
maxlen = max(len(seq) for seq in X_train_sequences)
```

This finds the longest sequence in X_train_sequences, which determines the number of time steps the model will process for each input during training.

Padding

That demonstrates how to perform padding on the tokenized sequences using the pad_sequences function. Padding is an essential preprocessing step in natural language processing (NLP) to ensure that all input sequences have the same length, making them compatible with machine learning models. The pad_sequences function takes the tokenized sequences (X_train_sequences) and ensures all sequences have the same length by padding them to the right (post-padding) with zeros. The length of each sequence is determined by maxlen, which represents the maximum sequence length found earlier. For example, if maxlen=10, a sequence like [1, 2, 3] will become [1, 2, 3, 0, 0, 0, 0, 0, 0, 0]. Machine learning models require inputs with consistent dimensions. Padding ensures that all sequences have the same length (maxlen), regardless of their original lengths.

Embedding

The code calculates the embedding input size, which is the total number of unique tokens (vocabulary size) in the training data. This value is critical when defining an embedding layer in a deep learning model, as it specifies the range of input indices that the embedding layer will handle. The +1 accounts for the zero-padding token as well. The embedding input size ensures that the embedding layer has enough capacity to map every token in the vocabulary (including the zero-padding token) to a dense vector representation. This allows the model to learn meaningful representations for all words in the dataset.

Implementation

The implementation is available at the file.
Model Initialization: The Sequential model is used to stack layers sequentially. Each layer's output serves as the input to the next layer.

Embedding Layer: Converts word indices from the input sequence into dense vector representations. The input_dim specifies the vocabulary size, output_dim determines the size of the embedding vectors, and input_length ensures all input sequences are the same length.

Bidirectional GRU Layer: A GRU processes the input sequence in both forward and backward directions, capturing context from both past and future words. The units parameter specifies the dimensionality of the GRU output.

Batch Normalization: Normalizes the activations from the GRU layer, ensuring faster convergence and more stable training by reducing internal covariate shift.

Dropout Layer: Drops 50% of the input units randomly during training to prevent overfitting and make the model more robust.

Dense Layer with ReLU Activation: A fully connected layer with 64 units and ReLU activation introduces non-linearity, enabling the model to learn complex patterns in the data.

Second Dropout Layer: Another dropout layer further regularizes the model, reducing the likelihood of overfitting.

Output Layer: The final dense layer outputs probabilities for 6 classes using the softmax activation function. Each unit corresponds to a class, and the outputs sum to 1, representing probabilities.

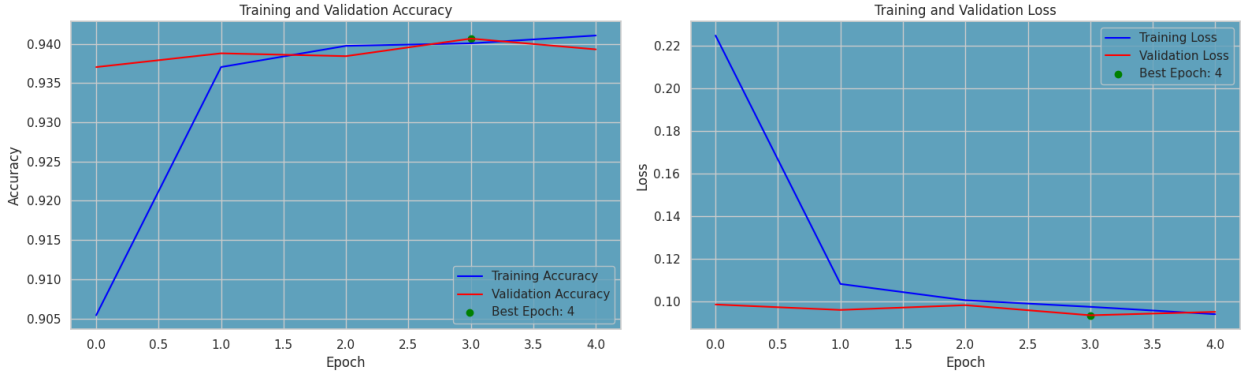
Compile

Adam Optimizer: An adaptive optimizer that adjusts the learning rate during training.
Categorical Crossentropy: A suitable loss function for multi-class classification tasks.
Accuracy Metric: Used to evaluate the model's performance during training and testing.

Model Train

We use 5 epochs as is shown on the output.

Training and Validation Accuracy & Loss



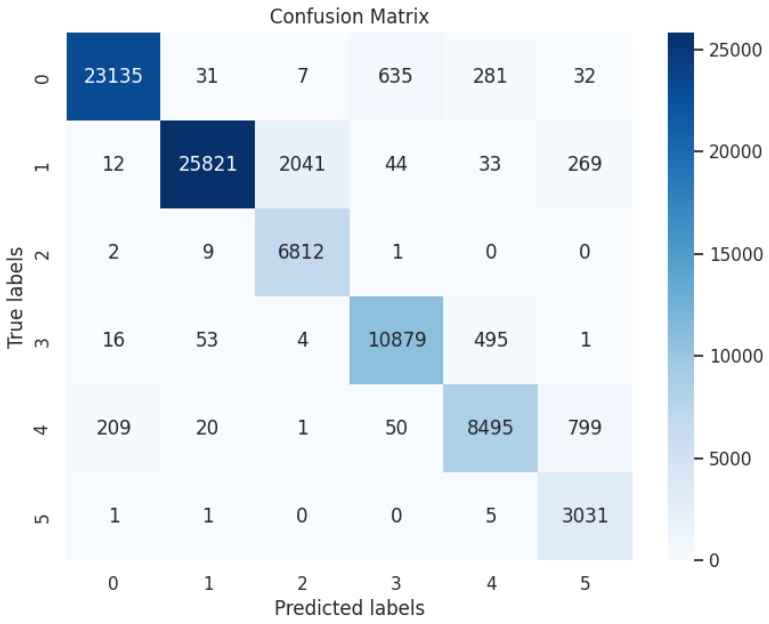
Left plot

The blue line represents the training accuracy, while the red line represents the validation accuracy. The training accuracy increases steadily, showing that the model is learning from the training data. By the 4th epoch, it reaches near-perfect accuracy. The validation accuracy improves slightly over the first few epochs and stabilizes. The best epoch is marked as Epoch 4 with the highest validation accuracy. This suggests that the model generalizes well to unseen data without significant overfitting. The convergence of the training and validation accuracy lines indicates that the model is not overfitting and is likely well-tuned.

Right Plot

The blue line represents the training loss, and the red line represents the validation loss.
Training loss decreases rapidly, indicating that the model quickly learns the patterns in the training data. Validation loss decreases initially and stabilizes, with the best epoch marked at Epoch 4, where the validation loss is at its minimum. The minimal gap between training and validation loss suggests the model is performing well without overfitting, as the validation loss remains close to the training loss throughout.
Best Epoch: The optimal model is at Epoch 4, where validation accuracy peaks and validation loss is minimized.

Confusion Matrix



The completed code generates a heatmap to visualize the confusion matrix, providing a clear representation of how the model's predicted labels align with the true labels. Each cell in the heatmap corresponds to the count of instances for a specific combination of predicted and true labels, with darker shades of blue indicating higher values. The diagonal cells show correctly classified instances, while off-diagonal cells reveal misclassifications. Emotion labels from the emotion_map are used as axis labels, making the visualization intuitive and easy to interpret. Additionally, the classification report offers a detailed summary of the model's performance for each class, including precision, recall, F1-score, and support. These metrics help assess how well the model predicts each emotion and identify areas where it struggles, such as distinguishing between similar categories. Together, the heatmap and classification report provide a comprehensive understanding of the model's strengths and weaknesses, highlighting both accurate predictions and areas for refinement.

The confusion matrix provides a detailed breakdown of the model's classification performance by comparing the true labels (rows) against the predicted labels (columns). The diagonal values

represent correctly classified instances, such as 23,135 for class 0 and 25,821 for class 1, indicating that the model is performing well for these classes. Misclassifications are represented by off-diagonal values, with notable examples including 635 instances of class 0 being misclassified as class 3 and 2,041 instances of class 1 being predicted as class 2. This suggests some overlap in the features between these classes, possibly due to similarities in the data.

Class-specific analysis shows that the model performs particularly well for classes 0, 1, and 3, which have high diagonal values and minimal confusion. However, there is notable misclassification between similar or overlapping classes, such as between class 1 (e.g., "joy") and class 2 (e.g., "love") or between class 4 (e.g., "fear") and class 5 (e.g., "surprise"). These misclassifications highlight areas where the model could be improved, possibly by increasing the representation of underrepresented classes or fine-tuning class-specific features.