

Group 1 - Twitter Emoji Prediction Deep Learning Project

Group Members:

- Safoora Akrami
- Lok Yin Wong
- Lily Gharacheh
- Shashi Singh
- Vrushank Sharma
- Avinash Sudireddy
- Ishika Fatwani

Step 1: Setup & Project Overview

Dataset Source:


We are using the [Twitter Emoji Prediction dataset](#) from Kaggle. It contains tweets labeled with emojis, used to train models for multi-class emoji prediction.

Objective: Build a deep learning model to predict the most appropriate emoji for a tweet.

Problem Type: Multi-class classification problem with 20 emoji categories.

Why This Matters: Improves social media analytics and sentiment understanding Real-world application of NLP and deep learning

```
1 !curl -L -o ./twitter-emoji-prediction.zip https://www.kaggle.com/api/v1/datasets/download/hariharasudhanas/twitter-emoji-prediction
2 !unzip -q twitter-emoji-prediction.zip
```

	% Total	% Received	% Xferd	Average	Speed	Time	Time	Time	Current
				Dload	Upload	Total	Spent	Left	Speed
	0	0	0	0	0	--:--:--	--:--:--	--:--:--	0
	100	3579k	100	3579k	0	0	5627k	0	--:--:--
									5627k

Library Installation

We install the latest versions of:

- `transformers` : for working with pretrained models like BERT.
- `keras` : for deep learning model building.

```
1 !pip install -q --upgrade transformers
2 !pip install -q --upgrade keras
```

	10.5/10.5 MB	52.3 MB/s	eta 0:00:00
	1.4/1.4 MB	11.3 MB/s	eta 0:00:00

Compatibility Settings

We explicitly set the environment variable `TF_USE_LEGACY_KERAS=1` to ensure compatibility between Keras and Hugging Face Transformers.

```
1 import os
2 os.environ['TF_USE_LEGACY_KERAS'] = '1'
```

Library Imports

We begin by importing essential libraries:


- **NumPy (np)**: For array and numerical computations.
- **Pandas (pd)**: For loading and managing tabular data.
- **TensorFlow (tf)**: Main framework used for building and training neural networks.
- **Matplotlib (plt)**: For plotting evaluation metrics like accuracy and loss.

```
1 # Import necessary libraries
2 import numpy as np
3 import pandas as pd
4 import tensorflow as tf
5 import matplotlib.pyplot as plt
6 from google.colab import files
```

✓ Check for GPU Availability

We check if a GPU is available using TensorFlow. If not, we fall back to using the CPU.

```
1 # Check if GPU is present
2
3 try:
4     device_name = tf.test.gpu_device_name()
5     if device_name != '/device:GPU:0':
6         raise SystemError('GPU device not found')
7     print('Found GPU at: {}'.format(device_name))
8 except:
9     device_name = '/cpu:0'
10    print('No GPU found. Using CPU instead.')
11
12 print(f"Using device {device_name}")
```

 No GPU found. Using CPU instead.
Using device /cpu:0

✓ Step 2: Load and Explore the Dataset

Data Info:

- ~70,000 labelled Tweets
- Each Tweet is labelled with 1 of 20 Emojis
- Input: Tweet text
- Target: Emoji class (label)

We load the training data from `Train.csv`, keeping only the relevant columns: tweet text and its corresponding label and then rename its columns to lowercase.

```
1 # Load train set
2 train_df = pd.read_csv('Train.csv', usecols=[1, 2])
3
4 train_df.columns = train_df.columns.str.lower()
5 train_df.head()
```



	text	label
0	Vacation wasted ! #vacation2017 #photobomb #ti...	0
1	Oh Wynwood, you're so funny! : @user #Wynwood ...	1
2	Been friends since 7th grade. Look at us now w...	2
3	This is what it looks like when someone loves ...	3
4	RT @user this white family was invited to a Bl...	3

Emoji Label Mapping Reads only columns 1 and 2 (ignoring column 0) Assigns custom column names: "emoji" and "label"

To interpret model predictions, we load `Mapping.csv`, which maps numeric labels to actual emojis.

```
1 # Load emoji mapping
2 mapping_df = pd.read_csv("Mapping.csv", usecols=[1, 2], names=["emoji", "label"], header=0)
3
4 mapping_df.head()
```



	emoji	label
0	😞	0
1	🤔	1
2	😞	2
3	😞	3
4	😞	4

Number of entries: 70,000 Columns: text (tweets) and label (emoji class) Data types: object for text, int64 for labels

Confirms there are no missing values, Finds and counts duplicate rows (there are 69), Checks for missing (null) values (there are none)

```
1 train_df.info()
2 print("Duplicated rows", train_df.duplicated().sum())
3 print("Null rows", train_df.isna().sum())
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70000 entries, 0 to 69999
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype  
---  -
0    text        70000 non-null   object  
1    label        70000 non-null   int64   
dtypes: int64(1), object(1)
memory usage: 1.1+ MB
Duplicated rows 69
Null rows text    0
      label      0
      dtype: int64

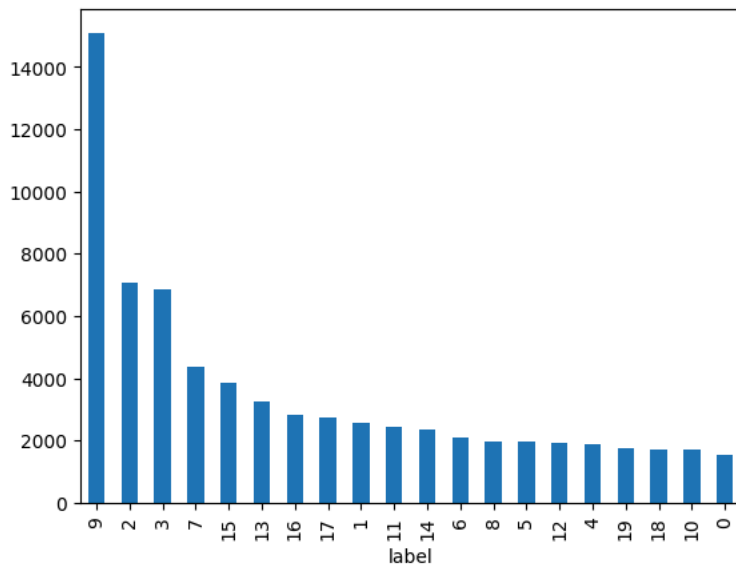
1 print(train_df.value_counts("label"))
2 train_df["label"].value_counts().plot(kind="bar")

```

```

label
9    15091
2     7076
3     6842
7     4363
15    3843
13    3250
16    2832
17    2751
1     2592
11    2434
14    2342
6     2083
8     1992
5     1977
12    1909
4     1878
19    1745
18    1722
10    1721
0     1557
Name: count, dtype: int64
<Axes: xlabel='label'>

```



Step 3: Data Preprocessing

Text Cleaning

In the data preprocessing part, we first drop the duplicated rows we identified before.

Then, we clean the text by removing mentions, URLs and hashtags. After that, the text is converted to lower case and stripped.

```

1 # Clean text
2 import re
3 def clean_text(text):
4     """Cleans a given text string by removing mentions, URLs, and hashtags.
5
6     This function performs the following operations:
7     - Removes Twitter-style mentions (e.g., @username)
8     - Removes URLs (e.g., http://example.com)
9     - Removes hashtags (e.g., #topic)
10    - Converts the text to lowercase
11    - Strips leading and trailing whitespace
12

```

```

13 Args:
14     text (str): The input text string to clean.
15
16 Returns:
17     str: The cleaned text.
18 """
19 text = re.sub(r"@w+", "", text)
20 text = re.sub(r"http\S+", "", text)
21 text = re.sub(r"#w+", "", text)
22 return text.lower().strip()
23
24 train_df.drop_duplicates(inplace=True)
25 train_df["clean_text"] = train_df["text"].apply(clean_text)
26 train_df.head()

```



	text	label	clean_text
0	Vacation wasted ! #vacation2017 #photobomb #ti...	0	vacation wasted ! @ port of...
1	Oh Wynwood, you're so funny! : @user #Wynwood ...	1	oh wynwood, you're so funny! : @ wynwood...
2	Been friends since 7th grade. Look at us now w...	2	been friends since 7th grade. look at us now w...
3	This is what it looks like when someone loves ...	3	this is what it looks like when someone loves ...
4	RT @user this white family was invited to a Bl...	3	rt this white family was invited to a black b...

Distribution Balancing

Since the distribution of labels is imbalance. We decided to keep only the top 9 labels with largest amount of samples.

The script performs the following

1. Filters the dataset to include only the specified target_labels
2. Downsamples each class to match the smallest class count
3. Shuffles the resulting balanced dataset
4. Visualizes the new class distribution with a bar chart

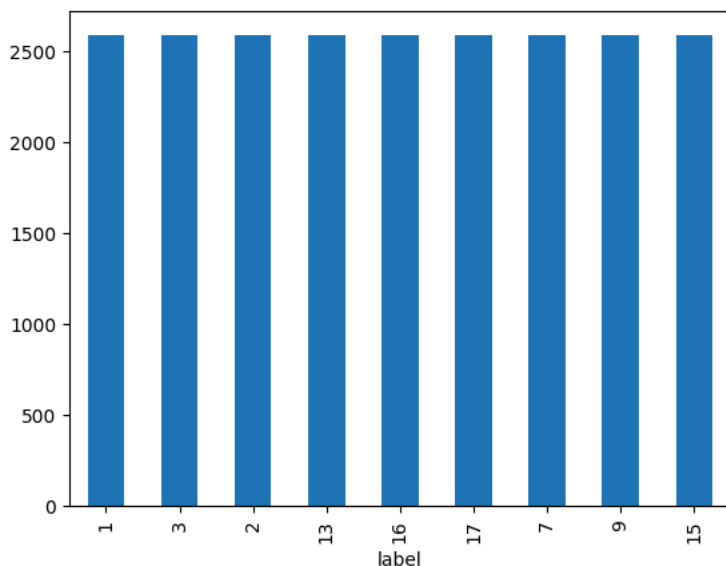
```

1 target_labels = [9, 2, 3, 7, 15, 13, 16, 17, 1]
2
3 filtered_df = train_df[train_df['label'].isin(target_labels)]
4
5 min_count = filtered_df['label'].value_counts().min()
6
7 balanced_df = pd.concat([
8     group.sample(n=min_count, random_state=1234)
9     for _, group in filtered_df.groupby('label')
10 ])
11
12 balanced_df = balanced_df.sample(frac=1, random_state=42).reset_index(drop=True)
13
14 balanced_df["label"].value_counts().plot(kind="bar")

```



<Axes: xlabel='label'>



Step 4: Train / Validation Split

Splits cleaned and labeled text data into training, validation, and test sets using stratified sampling.

The script performs the following:

1. Extracts texts and labels from a preprocessed DataFrame to a numpy array.
2. Maps string labels to integer indices.
3. Splits the dataset into training, validation, and test sets using a fixed seed and stratification.

```
1 from sklearn.model_selection import train_test_split
2
3 seed = 1234
4
5 texts = balanced_df['clean_text'].to_numpy()
6 labels = balanced_df['label'].to_numpy()
7
8 original_labels = sorted(list(set(labels)))
9 label2idx = {label: idx for idx, label in enumerate(original_labels)}
10
11 # Map labels
12 labels_mapped = np.array([label2idx[label] for label in labels])
13
14 train_test_texts, val_texts, train_test_labels, val_labels = train_test_split(
15     texts, labels_mapped, test_size=0.2, random_state=seed, stratify=labels
16 )
17
18 train_texts, test_texts, train_labels, test_labels = train_test_split(
19     train_test_texts, train_test_labels, test_size=0.1, random_state=seed, stratify=train_test_labels
20 )
21
22 print(f"Train size: {len(train_texts)}")
23 print(f"Test size: {len(test_texts)}")
24 print(f"Validation size: {len(val_texts)}")
```

```
🔗 Train size: 16789
    Test size: 1866
    Validation size: 4664
```

▼ Step 5: Encode texts

We chose to encode the texts in two ways - BertTokenizer and Vectorizer with gloVe Embeddings. This is the first method.

The script performs the following

1. Load pretrained Bert tokenizer
2. Define a function that tokenize the texts and create a tf dataset for BERT-based models
3. Call tokenize for train, val, and test set respectively and batch them

```
1 from transformers import BertTokenizerFast
2
3 bert_tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
4
5 MAX_LEN = 100
6 BATCH_SIZE = 32
7
8 def tokenize(texts, labels, max_len=100):
9     """Tokenizes input texts and creates a TensorFlow dataset for BERT-based models.
10
11     This function uses a pretrained BERT tokenizer to convert raw text inputs into
12     token ID and attention mask pairs. The resulting tokenized inputs and associated
13     labels are formatted as a `tf.data.Dataset`.
14
15     Args:
16         texts (list or np.ndarray): A list or array of input text strings to tokenize.
17         labels (list or np.ndarray): Corresponding labels for each input text.
18         max_len (int, optional): The maximum sequence length for tokenization. Defaults to 100.
19
20     Returns:
21         tf.data.Dataset: A dataset of tokenized inputs and labels, where each element is a tuple:
22             ({
23                 "input_ids": tf.Tensor,
24                 "attention_mask": tf.Tensor
25             }, label)
26     """
27     encodings = bert_tokenizer(
28         texts,
29         truncation=True,
30         padding='max_length',
31         max_length=max_len,
32     )
33     return tf.data.Dataset.from_tensor_slices(({
34         "input_ids": encodings["input_ids"],
35         "attention_mask": encodings["attention_mask"]
```

```

36     }, labels))
37
38 # Create train and val datasets
39 bert_train_ds = tokenize(train_texts.tolist(), train_labels, max_len=MAX_LEN).cache().batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
40 bert_test_ds = tokenize(test_texts.tolist(), test_labels, max_len=MAX_LEN).cache().batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
41 bert_val_ds = tokenize(val_texts.tolist(), val_labels, max_len=MAX_LEN).cache().batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

```



/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:

The secret 'HF_TOKEN' does not exist in your Colab secrets.

To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret in your Google Colab and restart your session. You will be able to reuse this secret in all of your notebooks.

Please note that authentication is recommended but still optional to access public models or datasets.

warnings.warn(

tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 1.42kB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 2.28MB/s]

tokenizer.json: 100% 466k/466k [00:00<00:00, 6.56MB/s]

config.json: 100% 570/570 [00:00<00:00, 9.50kB/s]

The following script save the tokenizer and clean text function into a class instance and dump into a pickle object. This makes preprocessing easier in Streamlit.

```

1 import pickle
2 import re
3
4 class TextProcessor:
5     """Processes and tokenizes text data for BERT-style models.
6
7     This class provides methods for cleaning raw text and converting it
8     into tokenized input suitable for transformer-based models such as BERT.
9
10    Attributes:
11        tokenizer: A Hugging Face tokenizer instance used for tokenization.
12    """
13    def __init__(self, tokenizer):
14        """Initializes the TextProcessor with a given tokenizer.
15
16        Args:
17            tokenizer: A Hugging Face tokenizer (e.g., BertTokenizer or BertTokenizerFast).
18        """
19        self.tokenizer = tokenizer
20
21    def clean_text(self, text):
22        """Cleans the input text by removing mentions, URLs, and hashtags.
23
24        Applies regex-based cleaning to strip out unwanted patterns and normalizes the text.
25
26        Args:
27            text (str): The raw input string.
28
29        Returns:
30            str: A cleaned and lowercased version of the input text.
31        """
32        text = re.sub(r"@w+", "", text)
33        text = re.sub(r"http\S+", "", text)
34        text = re.sub(r"#w+", "", text)
35        return text.lower().strip()
36
37    def __call__(self, text, max_len=100):
38        """Cleans and tokenizes the input text, returning a TensorFlow dataset.
39
40        This method makes the class callable. It performs cleaning and tokenization,
41        and returns a dataset with input IDs and attention masks.
42
43        Args:
44            text (str): The input text string to process.
45            max_len (int, optional): Maximum sequence length for padding/truncation. Defaults to 100.
46
47        Returns:
48            tf.data.Dataset: A dataset with a single element containing:
49                {
50                    "input_ids": tf.Tensor,
51                    "attention_mask": tf.Tensor
52                }
53        """
54        cleaned_text = self.clean_text(text)
55        encodings = self.tokenizer(
56            [cleaned_text], # Wrap in a list as tokenizer expects list of strings
57            truncation=True,
58            padding='max_length',
59            max_length=max_len,
60            return_tensors='tf'
61        )

```

```

62
63     return tf.data.Dataset.from_tensor_slices({
64         "input_ids": encodings["input_ids"],
65         "attention_mask": encodings["attention_mask"]
66     })
67
68 text_processor = TextProcessor(bert_tokenizer)
69
70 with open('bert_text_processor.pkl', 'wb') as f:
71     pickle.dump(text_processor, f)
72
73 files.download("bert_text_processor.pkl")

```



Step 6: Build Model

The following section lists various models to approach this problem.

The section is divided into two main stream of models - BERT-based and gloVe embedding-based.

Utils Functions

Computes balanced class weights for use in model training.

This script calculates weights to address class imbalance in the training data using the `balanced` mode from scikit-learn. The resulting weights can be used during model training to give higher importance to underrepresented classes.

```

1 from sklearn.utils.class_weight import compute_class_weight
2 class_weights = compute_class_weight(
3     class_weight='balanced',
4     classes=np.unique(train_labels),
5     y=train_labels
6 )
7
8 class_weight_dict = {i: w for i, w in enumerate(class_weights)}
9 print("Class Weights:", class_weight_dict)
10
11 num_classes = len(set(labels))

```

```

Class Weights: {0: np.float64(0.9997022746218888), 1: np.float64(0.9997022746218888), 2: np.float64(0.9997022746218888), 3: np.float64(1.

```

```

1 def plot_history(history):
2     """Plots training and validation accuracy and loss from a Keras History object.
3
4     This function visualizes the accuracy and loss over training epochs using data
5     from a Keras 'History' object returned by 'model.fit()'. It produces two subplots:
6     one for accuracy and one for loss, each showing training and validation curves.
7
8     Args:
9         history (tf.keras.callbacks.History): The history object returned by model training.
10
11     Raises:
12         ValueError: If the history object is None or does not contain expected keys.
13
14     Example:
15         history = model.fit(...)
16         plot_history(history)
17     """
18
19     if not history:
20         print("No training history provided.")
21         return
22
23     acc = history.history.get('accuracy')
24     val_acc = history.history.get('val_accuracy')
25     loss = history.history.get('loss')
26     val_loss = history.history.get('val_loss')
27     epochs = range(1, len(acc) + 1)
28
29     plt.figure(figsize=(14, 5))
30
31     # Accuracy
32     plt.subplot(1, 2, 1)
33     plt.plot(epochs, acc, 'bo-', label='Training Accuracy')
34     plt.plot(epochs, val_acc, 'ro-', label='Validation Accuracy')
35     plt.title('Training and Validation Accuracy')
36     plt.xlabel('Epochs')
37     plt.ylabel('Accuracy')
38     plt.legend()
39
40

```

```

39 # Loss
40 plt.subplot(1, 2, 2)
41 plt.plot(epochs, loss, 'bo-', label='Training Loss')
42 plt.plot(epochs, val_loss, 'ro-', label='Validation Loss')
43 plt.title('Training and Validation Loss')
44 plt.xlabel('Epochs')
45 plt.ylabel('Loss')
46 plt.legend()
47
48
49 plt.tight_layout()
50 plt.show()
51
52 class DotDict:
53     """Converts a dictionary to an object with dot-accessible attributes."""
54     def __init__(self, dictionary):
55         for key, value in dictionary.items():
56             setattr(self, key, value)
57
58 def merge_histories(h1, h2):
59     """Merges the history objects of two model training runs.
60
61     This function assumes that both `h1` and `h2` are Keras History objects
62     and merges their metrics by concatenating the lists.
63
64     Args:
65         h1 (tf.keras.callbacks.History): History from the first training phase.
66         h2 (tf.keras.callbacks.History): History from the second training phase.
67
68     Returns:
69         DotDict: A dot-accessible dictionary with the merged 'history' key.
70     """
71     merged = {}
72     for key in h1.history:
73         merged[key] = h1.history[key] + h2.history[key]
74     return DotDict({"history": merged})

```

✓ Bert transformer

RAM: 7.2GB GPU RAM: 4.0GB

BERT (Bidirectional Encoder Representations from Transformers) is a powerful pre-trained language model developed by Google that understands language context by reading text bidirectionally. In this project, we explored multiple ways of leveraging BERT for text classification to improve accuracy, generalization, and feature extraction.

```

1 from tensorflow.keras import backend as K
2 K.clear_session()
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

✓ Fine-Tuned BERT (TFBertForSequenceClassification)

Why: This approach adapts the entire BERT model, including its internal attention weights, to our specific classification task.

Advantage: Since BERT was trained on a massive corpus (e.g., Wikipedia and BooksCorpus), fine-tuning allows us to benefit from its language understanding while customizing it to our dataset.

Use Case: Ideal for quick and effective performance with minimal custom architecture.

```

1 from transformers import TFBertForSequenceClassification
2
3 with tf.device(device_name):
4     bert_fine_tune_model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=num_classes)
5
6     bert_fine_tune_model.compile(
7         optimizer=tf.keras.optimizers.Adam(learning_rate=3e-5),
8         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
9         metrics=['accuracy']
10    )
11
12 bert_fine_tune_model.summary()
13
14 history = bert_fine_tune_model.fit(
15     bert_train_ds,
16     validation_data=bert_val_ds,
17     class_weight=class_weight_dict,
18     batch_size=BATCH_SIZE,
19     epochs=4,
20     callbacks=[tf.keras.callbacks.EarlyStopping(patience=2, restore_best_weights=True)]

```



```

21 )
22
23 plot_history(history)
24 bert_fine_tune_model.save_pretrained("fine_tuned_bert")
25

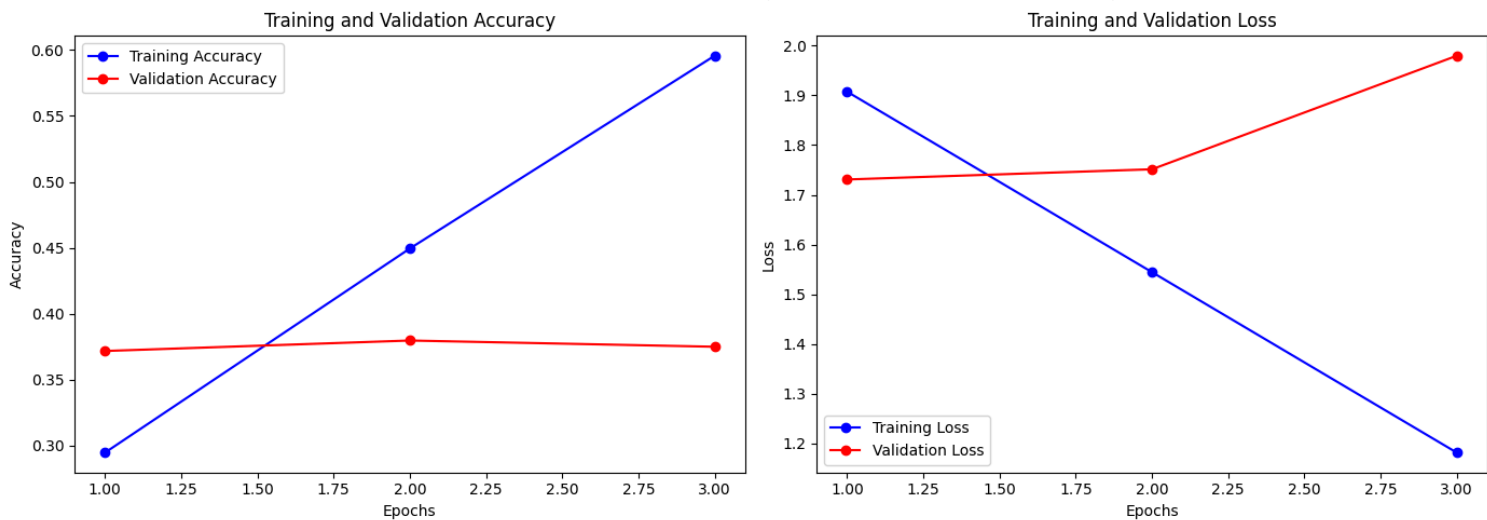
```

🔄 All PyTorch model weights were used when initializing TFBertForSequenceClassification.

Some weights or buffers of the TF 2.0 model TFBertForSequenceClassification were not initialized from the PyTorch model and are newly initialized: ['classifier.weight', 'classifier.bias']
 You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
 Model: "tf_bert_for_sequence_classification"

Layer (type)	Output Shape	Param #
=====		
bert (TFBertMainLayer)	multiple	109482240
dropout_132 (Dropout)	multiple	0 (unused)
classifier (Dense)	multiple	6921
=====		
Total params: 109489161 (417.67 MB)		
Trainable params: 109489161 (417.67 MB)		
Non-trainable params: 0 (0.00 Byte)		

Epoch 1/4
 525/525 [=====] - 230s 333ms/step - loss: 1.9072 - accuracy: 0.2947 - val_loss: 1.7308 - val_accuracy: 0.3718
 Epoch 2/4
 525/525 [=====] - 157s 298ms/step - loss: 1.5445 - accuracy: 0.4495 - val_loss: 1.7513 - val_accuracy: 0.3797
 Epoch 3/4
 525/525 [=====] - 157s 299ms/step - loss: 1.1818 - accuracy: 0.5956 - val_loss: 1.9795 - val_accuracy: 0.3750



Accuracy Trend:

- Training Accuracy increased rapidly, reaching ~60% by epoch 3
- Validation Accuracy stayed flat around 38%, showing no improvement after epoch 1
- Indicates overfitting — model learns the training data well but doesn't generalize

Loss Trend:

- Training Loss decreased significantly from ~1.9 to ~1.2
- Validation Loss slightly increased, ending above 1.9
- Confirms that the model is memorizing the training data rather than learning meaningful patterns

Despite high training performance, the gap between training and validation shows poor generalization

The model is too complex for the current data size, or the data itself has label noise or ambiguity

```

1 !zip -r fine_tuned_bert.zip fine_tuned_bert
2 files.download("fine_tuned_bert.zip")

```

✓ Bert with Classification Layer

RAM: 7.1GB GPU RAM: 4.1GB

BERT + Custom Classification Layers (CLS Token with Dense Layers)

Why: Instead of using BERT's default classification head, this approach extracts the [CLS] token embedding and passes it through a custom multi-layer neural network.

Advantage:

More flexibility in architecture design (e.g., adding batch normalization, regularization, and dropout).

Better control over model capacity and regularization, especially for domain-specific data.

Use Case: When you want to fine-tune BERT but also apply deep learning best practices to tailor the classifier head.

```
1 from transformers import TFBertModel
2 from tensorflow.keras.layers import Input, Dense, Dropout, Lambda, BatchNormalization, Layer
3 from tensorflow.keras.models import Model
4
5 class BertCLS(Layer):
6     """Custom Keras layer to extract the [CLS] token embedding from a BERT model.
7
8     This layer wraps a pretrained BERT model and returns the output corresponding to the
9     [CLS] token, which is typically used for classification tasks.
10
11     Attributes:
12         bert: A pretrained Hugging Face BERT model (e.g., TFBertModel).
13     """
14     def __init__(self, bert_model, **kwargs):
15         super().__init__(**kwargs)
16         self.bert = bert_model
17
18     def call(self, inputs):
19         input_ids, attention_mask = inputs
20         outputs = self.bert(input_ids, attention_mask=attention_mask)
21         return outputs.last_hidden_state[:, 0, :] # [CLS] token output
22
23 def get_bert_with_class_layer_model(num_classes):
24     """Builds a BERT-based text classification model with custom classification layers.
25
26     This function constructs a TensorFlow Keras model that uses a pretrained BERT model
27     as a base and adds custom fully connected layers for classification. The [CLS] token
28     embedding is extracted via a custom 'BertCLS' layer.
29
30     Architecture:
31         - BERT model ('bert-base-uncased') for contextual encoding
32         - [CLS] token extraction via 'BertCLS'
33         - Fully connected layers with dropout, batch normalization, and ReLU activations
34         - Final softmax output layer for multi-class classification
35
36     Args:
37         num_classes (int): Number of target classes for classification.
38
39     Returns:
40         tf.keras.Model: A compiled Keras model ready for training.
41
42     Example:
43         model = get_bert_with_class_layer_model(num_classes=5)
44         model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
45     """
46     bert = TFBertModel.from_pretrained("bert-base-uncased")
47     input_ids = Input(shape=(MAX_LEN,), dtype=tf.int32, name='input_ids')
48     attention_mask = Input(shape=(MAX_LEN,), dtype=tf.int32, name='attention_mask')
49
50     x = BertCLS(bert)([input_ids, attention_mask])
51
52     x = Dropout(0.3)(x)
53     x = Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
54     x = BatchNormalization()(x)
55     x = Dropout(0.3)(x)
56     x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
57     x = BatchNormalization()(x)
58     x = Dense(128, activation='relu')(x)
59     x = Dropout(0.3)(x)
60
61     output = Dense(num_classes, activation='softmax')(x)
62
63     model = Model(inputs=[input_ids, attention_mask], outputs=output)
64     return model
```

Trains a BERT-based classification model using a custom classifier head.

This script compiles and trains a BERT model with learning rate scheduling, early stopping, and learning rate reduction on plateau. It also plots training history and saves the trained model weights.

```
1 num_train_samples = len(train_labels)
2 epochs = 10
3 total_steps = (num_train_samples // BATCH_SIZE) * epochs
4
5 with tf.device(device_name):
6     bert_with_class_layer_model = get_bert_with_class_layer_model(num_classes)
```

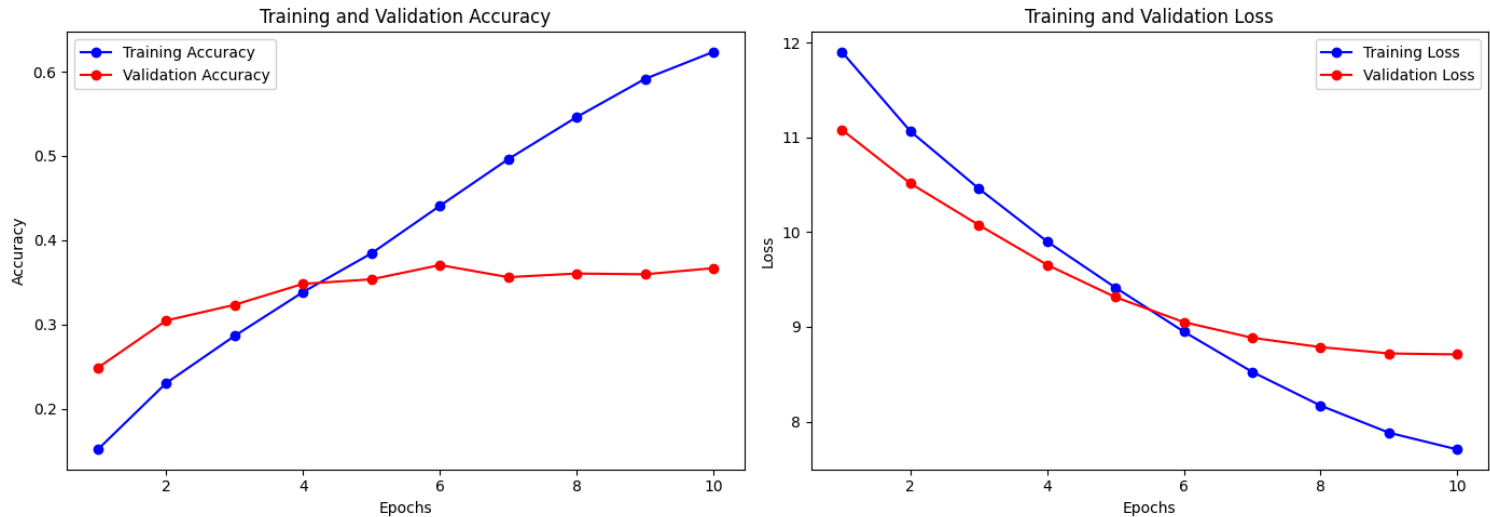
```
7
8 lr_schedule = tf.keras.optimizers.schedules.PolynomialDecay(
9     initial_learning_rate=4e-5,
10    decay_steps=total_steps,
11    end_learning_rate=1e-6
12 )
13
14 bert_with_class_layer_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr_schedule), loss='sparse_categorical_crossentropy')
15 bert_with_class_layer_model.summary()
16
17 history = bert_with_class_layer_model.fit(
18     bert_train_ds,
19     validation_data=bert_val_ds,
20     class_weight=class_weight_dict,
21     epochs=epochs,
22     batch_size=BATCH_SIZE,
23     callbacks=[tf.keras.callbacks.EarlyStopping(patience=2, restore_best_weights=True), tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', patience=5)]
24 )
25 plot_history(history)
26 bert_with_class_layer_model.save_weights("bert_with_class_layer_model.weights.h5")
27
```

Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFBertModel: ['cls.predictions.transform.dense.weight', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.dense.bias'] - This IS expected if you are initializing TFBertModel from a PyTorch model trained on another task or with another architecture (e.g. initializing a TFBertForSequenceClassification model from a PyTorch model trained on another task) - This IS NOT expected if you are initializing TFBertModel from a PyTorch model that you expect to be exactly identical (e.g. initializing a TFBertForSequenceClassification model from a PyTorch model trained on the same task) All the weights of TFBertModel were initialized from the PyTorch model. If your task is similar to the task the model of the checkpoint was trained on, you can already use TFBertModel for predictions without further training. Model: "model_7"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	[]
attention_mask (InputLayer)	[(None, 100)]	0	[]
bert_cls_1 (BertCLS)	(None, 768)	1094822	['input_ids[0][0]', 'attention_mask[0][0]']
dropout_92 (Dropout)	(None, 768)	0	['bert_cls_1[0][0]']
dense_29 (Dense)	(None, 512)	393728	['dropout_92[0][0]']
batch_normalization_9 (Batch Normalization)	(None, 512)	2048	['dense_29[0][0]']
dropout_93 (Dropout)	(None, 512)	0	['batch_normalization_9[0][0]']
dense_30 (Dense)	(None, 256)	131328	['dropout_93[0][0]']
batch_normalization_10 (Batch Normalization)	(None, 256)	1024	['dense_30[0][0]']
dense_31 (Dense)	(None, 128)	32896	['batch_normalization_10[0][0]']
dropout_94 (Dropout)	(None, 128)	0	['dense_31[0][0]']
dense_32 (Dense)	(None, 9)	1161	['dropout_94[0][0]']

Total params: 110044425 (419.79 MB)
Trainable params: 110042889 (419.78 MB)
Non-trainable params: 1536 (6.00 KB)

Epoch 1/10
WARNING:tensorflow:Gradients do not exist for variables ['tf_bert_model_1/bert/pooler/dense/kernel:0', 'tf_bert_model_1/bert/pooler/dense/bias:0'] when minimizing the loss. If you're running an evaluation, this warning can be safely ignored. If you're running a training process, this warning indicates that your training data is not representative of your evaluation data.
WARNING:tensorflow:Gradients do not exist for variables ['tf_bert_model_1/bert/pooler/dense/kernel:0', 'tf_bert_model_1/bert/pooler/dense/bias:0'] when minimizing the loss. If you're running an evaluation, this warning can be safely ignored. If you're running a training process, this warning indicates that your training data is not representative of your evaluation data.
525/525 [=====] - 214s 315ms/step - loss: 11.9040 - accuracy: 0.1515 - val_loss: 11.0821 - val_accuracy: 0.2485 - lr: 3.6100e-05
Epoch 2/10
525/525 [=====] - 155s 295ms/step - loss: 11.0672 - accuracy: 0.2304 - val_loss: 10.5159 - val_accuracy: 0.3047 - lr: 3.2193e-05
Epoch 3/10
525/525 [=====] - 152s 289ms/step - loss: 10.4616 - accuracy: 0.2864 - val_loss: 10.0767 - val_accuracy: 0.3233 - lr: 2.8285e-05
Epoch 4/10
525/525 [=====] - 156s 297ms/step - loss: 9.9021 - accuracy: 0.3384 - val_loss: 9.6556 - val_accuracy: 0.3482 - lr: 2.4378e-05
Epoch 5/10
525/525 [=====] - 157s 299ms/step - loss: 9.4149 - accuracy: 0.3844 - val_loss: 9.3141 - val_accuracy: 0.3536 - lr: 2.0470e-05
Epoch 6/10
525/525 [=====] - 161s 306ms/step - loss: 8.9495 - accuracy: 0.4408 - val_loss: 9.0502 - val_accuracy: 0.3705 - lr: 1.6563e-05
Epoch 7/10
525/525 [=====] - 163s 310ms/step - loss: 8.5246 - accuracy: 0.4962 - val_loss: 8.8850 - val_accuracy: 0.3561 - lr: 1.2655e-05
Epoch 8/10
525/525 [=====] - 157s 299ms/step - loss: 8.1686 - accuracy: 0.5462 - val_loss: 8.7863 - val_accuracy: 0.3604 - lr: 8.7479e-06
Epoch 9/10
525/525 [=====] - 152s 290ms/step - loss: 7.8827 - accuracy: 0.5915 - val_loss: 8.7198 - val_accuracy: 0.3596 - lr: 4.8405e-06
Epoch 10/10
525/525 [=====] - 152s 290ms/step - loss: 7.7056 - accuracy: 0.6236 - val_loss: 8.7090 - val_accuracy: 0.3669 - lr: 1.0000e-06



Accuracy Trend:

- Training Accuracy increased steadily, reaching ~63% by epoch 10
- Validation Accuracy improved early on, peaking around 38%, then plateaued
- This indicates the model learned well, but generalization stopped improving after epoch 5

Loss Trend:

- Training Loss dropped consistently from ~12 to ~7.5
- Validation Loss decreased at first, then stabilized around 8.8, showing early convergence
- No major overfitting, but also no further gain in validation performance

Model is learning effectively, but validation performance plateaus early

```
1 files.download('bert_with_class_layer_model.weights.h5')
```



✓ Bidirectional LSTM with Bert Embeddings

RAM: 7.1GB GPU RAM: 4.1GB

BERT + Bidirectional LSTM (BERT as Feature Extractor)

Why: This setup uses BERT to generate contextualized token embeddings, then processes them using a BiLSTM layer to capture sequential dependencies.

Advantage:

- Sequential modeling: LSTM captures positional and temporal patterns, which BERT doesn't explicitly model.
- Fine-grained context: BiLSTM can enhance downstream task performance by re-interpreting BERT's embeddings with temporal dynamics.

Use Case: Useful for tasks where order and flow of tokens are crucial (e.g., sentiment analysis, complex linguistic structures).

```
1 from transformers import TFBertModel
2 from tensorflow.keras.layers import Input, Bidirectional, LSTM, Dense, Dropout, Lambda, BatchNormalization
3 from tensorflow.keras.models import Model
4
5 class Bert(Layer):
6     """Custom Keras layer to extract sequence output from a BERT model.
7
8     This layer wraps a pretrained Hugging Face BERT model and returns the
9     full sequence output ('last_hidden_state'), which includes contextual
10    embeddings for each token in the input.
11
12    Attributes:
13        bert: A pretrained Hugging Face BERT model (e.g., TFBertModel).
14    """
15    def __init__(self, bert_model, **kwargs):
16        super().__init__(name="bert_sequence_output", **kwargs)
17        self.bert = bert_model
18
19    def call(self, inputs):
20        input_ids, attention_mask = inputs
21        outputs = self.bert(input_ids, attention_mask=attention_mask)
22        return outputs.last_hidden_state
23
24 def get_bidirectional_lstm_bert_model(num_labels):
25     """Builds a BERT + Bidirectional LSTM model for text classification.
26
27     This model uses a frozen BERT encoder to extract token-level embeddings and passes
28     them through a Bidirectional LSTM followed by fully connected layers for
29     classification.
30
31    Architecture:
32        - Pretrained BERT (frozen weights)
33        - Bidirectional LSTM (128 units in each direction)
34        - Dense layers with dropout and batch normalization
35        - Softmax output for multi-class classification
36
37    Args:
38        num_labels (int): Number of output classes.
39
40    Returns:
41        tf.keras.Model: A compiled Keras model ready for training.
42    """
43    # Load base BERT model (no classification head)
```

```

44 bert_model = TFBertModel.from_pretrained('bert-base-uncased')
45 bert_model.trainable = False
46
47 # Input layers
48 input_ids = Input(shape=(MAX_LEN,), dtype=tf.int32, name='input_ids')
49 attention_mask = Input(shape=(MAX_LEN,), dtype=tf.int32, name='attention_mask')
50
51 x = Bert(bert_model)([input_ids, attention_mask]) # shape: (batch_size, MAX_LEN, 768)
52
53 x = Bidirectional(LSTM(128, return_sequences=False))(x) # shape: (batch_size, 256)
54
55 # Dense + Dropout + BatchNorm head
56 x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
57 x = BatchNormalization()(x)
58 x = Dropout(0.3)(x)
59
60 x = Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
61 x = BatchNormalization()(x)
62 x = Dropout(0.3)(x)
63
64 output = Dense(num_labels, activation='softmax')(x)
65
66 return Model(inputs=[input_ids, attention_mask], outputs=output)

```

Trains a two-phase BiLSTM-BERT model with optional fine-tuning.

This script:

- Initializes a frozen BERT + BiLSTM model.
- Trains it with BERT frozen for 3 epochs.
- Unfreezes BERT and continues fine-tuning.
- Merges training histories.
- Plots performance and saves final weights.

```

1 with tf.device(device_name):
2
3     bert_bi_lstm_model = get_bidirectional_lstm_bert_model(num_classes)
4
5     bert_bi_lstm_model.compile(
6         optimizer=tf.keras.optimizers.Adam(learning_rate=3e-5),
7         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
8         metrics=['accuracy']
9     )
10    bert_bi_lstm_model.summary()
11
12    history1 = bert_bi_lstm_model.fit(
13        bert_train_ds,
14        validation_data=bert_val_ds,
15        class_weight=class_weight_dict,
16        epochs=3,
17        callbacks=[tf.keras.callbacks.EarlyStopping(patience=2, restore_best_weights=True)]
18    )
19
20    bert_bi_lstm_model.get_layer("bert_sequence_output").bert.trainable = True
21
22    bert_bi_lstm_model.compile(
23        optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5),
24        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
25        metrics=['accuracy']
26    )
27    bert_bi_lstm_model.summary()
28
29    history2 = bert_bi_lstm_model.fit(
30        bert_train_ds,
31        validation_data=bert_val_ds,
32        class_weight=class_weight_dict,
33        initial_epoch=3,
34        epochs=10,
35        callbacks=[tf.keras.callbacks.EarlyStopping(patience=2, restore_best_weights=True)]
36    )
37
38    history = merge_histories(history1, history2)
39
40    plot_history(history)
41    bert_bi_lstm_model.save_weights("bidirectional_lstm_bert_model.weights.h5")

```



Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: `pip install huggingface_hub`
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance, install the package with: `pip install huggingface_hub`

model.safetensors: 100% 440M/440M [00:07<00:00, 67.6MB/s]

Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFBertModel: ['cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.weight', 'cls.predictions.bias'] - This IS expected if you are initializing TFBertModel from a PyTorch model trained on another task or with another architecture (e.g. initializing a TFBertForSequenceClassification model from a PyTorch model trained on a text classification task). All the weights of TFBertModel were initialized from the PyTorch model. If your task is similar to the task the model of the checkpoint was trained on, you can already use TFBertModel for predictions without further training.

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	[]
attention_mask (InputLayer)	[(None, 100)]	0	[]
bert_sequence_output (Bert)	(None, 100, 768)	1094822	['input_ids[0][0]', 'attention_mask[0][0]']
bidirectional (Bidirectional)	(None, 256)	918528	['bert_sequence_output[0][0]']
dense (Dense)	(None, 256)	65792	['bidirectional[0][0]']
batch_normalization (Batch Normalization)	(None, 256)	1024	['dense[0][0]']
dropout_37 (Dropout)	(None, 256)	0	['batch_normalization[0][0]']
dense_1 (Dense)	(None, 128)	32896	['dropout_37[0][0]']
batch_normalization_1 (Batch Normalization)	(None, 128)	512	['dense_1[0][0]']
dropout_38 (Dropout)	(None, 128)	0	['batch_normalization_1[0][0]']
dense_2 (Dense)	(None, 9)	1161	['dropout_38[0][0]']

=====
Total params: 110502153 (421.53 MB)
Trainable params: 1019145 (3.89 MB)
Non-trainable params: 109483008 (417.64 MB)

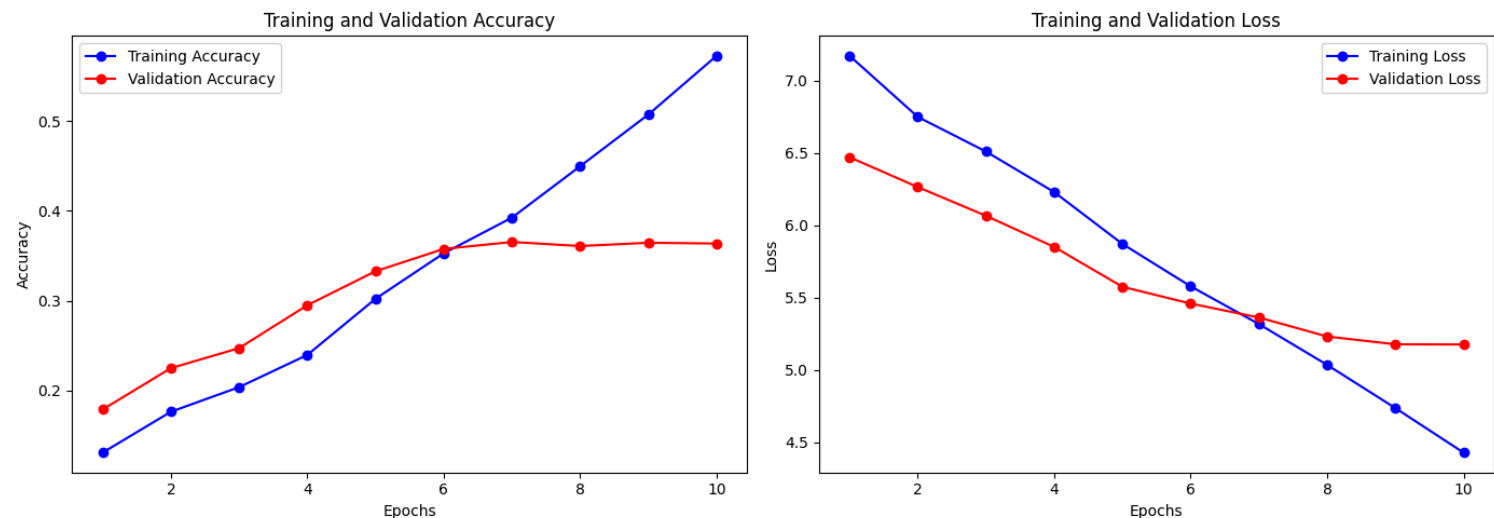
Epoch 1/3
525/525 [=====] - 107s 158ms/step - loss: 7.1729 - accuracy: 0.1305 - val_loss: 6.4719 - val_accuracy: 0.1786
Epoch 2/3
525/525 [=====] - 74s 142ms/step - loss: 6.7501 - accuracy: 0.1761 - val_loss: 6.2652 - val_accuracy: 0.2247
Epoch 3/3
525/525 [=====] - 75s 142ms/step - loss: 6.5107 - accuracy: 0.2035 - val_loss: 6.0669 - val_accuracy: 0.2470
Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	[]
attention_mask (InputLayer)	[(None, 100)]	0	[]
bert_sequence_output (Bert)	(None, 100, 768)	1094822	['input_ids[0][0]', 'attention_mask[0][0]']
bidirectional (Bidirectional)	(None, 256)	918528	['bert_sequence_output[0][0]']
dense (Dense)	(None, 256)	65792	['bidirectional[0][0]']
batch_normalization (Batch Normalization)	(None, 256)	1024	['dense[0][0]']
dropout_37 (Dropout)	(None, 256)	0	['batch_normalization[0][0]']
dense_1 (Dense)	(None, 128)	32896	['dropout_37[0][0]']
batch_normalization_1 (Batch Normalization)	(None, 128)	512	['dense_1[0][0]']
dropout_38 (Dropout)	(None, 128)	0	['batch_normalization_1[0][0]']
dense_2 (Dense)	(None, 9)	1161	['dropout_38[0][0]']

=====
Total params: 110502153 (421.53 MB)
Trainable params: 110501385 (421.53 MB)
Non-trainable params: 768 (3.00 KB)

Epoch 4/10
WARNING:tensorflow:Gradients do not exist for variables ['tf_bert_model/bert/pooler/dense/kernel:0', 'tf_bert_model/bert/pooler/dense/bias:0'] when minimizing the loss. If you're using tf.nn.softmax_cross_entropy_with_logits, you should use tf.nn.softmax_cross_entropy_with_logits_v2 instead.
WARNING:tensorflow:Gradients do not exist for variables ['tf_bert_model/bert/pooler/dense/kernel:0', 'tf_bert_model/bert/pooler/dense/bias:0'] when minimizing the loss. If you're using tf.nn.softmax_cross_entropy_with_logits, you should use tf.nn.softmax_cross_entropy_with_logits_v2 instead.
525/525 [=====] - 232s 355ms/step - loss: 6.2304 - accuracy: 0.2394 - val_loss: 5.8513 - val_accuracy: 0.2948
Epoch 5/10

```
Epoch 5/10
525/525 [=====] - 162s 309ms/step - loss: 5.8719 - accuracy: 0.3020 - val_loss: 5.5761 - val_accuracy: 0.3328
Epoch 6/10
525/525 [=====] - 166s 316ms/step - loss: 5.5791 - accuracy: 0.3528 - val_loss: 5.4602 - val_accuracy: 0.3576
Epoch 7/10
525/525 [=====] - 163s 310ms/step - loss: 5.3184 - accuracy: 0.3925 - val_loss: 5.3637 - val_accuracy: 0.3654
Epoch 8/10
525/525 [=====] - 171s 325ms/step - loss: 5.0362 - accuracy: 0.4499 - val_loss: 5.2326 - val_accuracy: 0.3608
Epoch 9/10
525/525 [=====] - 166s 316ms/step - loss: 4.7377 - accuracy: 0.5075 - val_loss: 5.1787 - val_accuracy: 0.3645
Epoch 10/10
525/525 [=====] - 166s 317ms/step - loss: 4.4297 - accuracy: 0.5730 - val_loss: 5.1773 - val_accuracy: 0.3636
```



```
/usr/local/lib/python3.11/dist-packages/transformers/generation/tf_utils.py:465: UserWarning: `seed_generator` is deprecated and will be removed in a future version.
warnings.warn("`seed_generator` is deprecated and will be removed in a future version.", UserWarning)
```

Accuracy trend:

- Training accuracy is increasing rapidly, indicating the model is learning from training data effectively.
- Validation accuracy peaks at Epoch 7, then flattens or declines slightly, even as training accuracy continues improving.
- This suggests the model starts overfitting after Epoch 7.

Loss trend:

- Training loss continues decreasing sharply.
- Validation loss plateaus after Epoch 8, despite training loss dropping further → another clear sign of overfitting.

```
1 files.download("bidirectional_lstm_bert_model.weights.h5")
```

✓ Glove Embeddings

GloVe (Global Vectors for Word Representation) is a widely used pre-trained word embedding model that captures the semantic meaning of words based on their global co-occurrence statistics in large text corpora. In this project, we use GloVe embeddings as the foundation for several neural models due to the following reasons:

1. Leverage Pre-trained Semantic Knowledge
 - GloVe is trained on massive text datasets (e.g., Wikipedia, Common Crawl).
 - Words that appear in similar contexts have similar vectors, allowing the model to generalize better with limited training data.
2. Improve Convergence and Stability
 - Pre-trained embeddings reduce the number of parameters the model must learn from scratch.
 - Leads to faster convergence, more stable training, and often better performance than training embeddings from random initialization.
3. Handle Small/Medium Datasets Better
 - When working with limited labeled data, training high-quality word embeddings from scratch is impractical.
 - GloVe provides a strong initialization, helping the model learn meaningful representations without overfitting.
4. Freeze to Reduce Overfitting
 - Preserve their semantic structure.
 - Reduce the number of trainable parameters.


```

1 !wget https://nlp.stanford.edu/data/glove.twitter.27B.zip
2 !unzip -q glove.twitter.27B.zip

```

```

--2025-05-31 20:35:24-- https://nlp.stanford.edu/data/glove.twitter.27B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu) ... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443 ... connected.
HTTP request sent, awaiting response ... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.twitter.27B.zip [following]
--2025-05-31 20:35:25-- https://downloads.cs.stanford.edu/nlp/data/glove.twitter.27B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu) ... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443 ... connected.
HTTP request sent, awaiting response ... 200 OK
Length: 1520408563 (1.4G) [application/zip]
Saving to: 'glove.twitter.27B.zip'

glove.twitter.27B.z 100%[=====>] 1.42G 5.01MB/s in 4m 45s

2025-05-31 20:40:09 (5.09 MB/s) - 'glove.twitter.27B.zip' saved [1520408563/1520408563]

```

```

1 def load_glove_embeddings(glove_file_path):
2     """Loads GloVe word embeddings from a file into a dictionary.
3
4     This function reads a GloVe-formatted file where each line contains a word
5     followed by its embedding vector components. It constructs a dictionary mapping
6     each word to its corresponding embedding vector.
7
8     Args:
9         glove_file_path (str): Path to the GloVe embedding file.
10
11     Returns:
12         dict: A dictionary where keys are words (str) and values are NumPy arrays
13              representing the embedding vectors (dtype=float32).
14
15     Example:
16         embeddings_index = load_glove_embeddings("glove.twitter.27B.100d.txt")
17         print(embeddings_index["happy"]) # Output: np.ndarray of shape (100,)
18     """
19     embeddings_index = {}
20     with open(glove_file_path, encoding='utf-8') as f:
21         for line in f:
22             values = line.strip().split()
23             word = values[0]
24             coefs = np.asarray(values[1:], dtype='float32')
25             embeddings_index[word] = coefs
26     return embeddings_index
27
28 glove_path = 'glove.twitter.27B.100d.txt'
29 embedding_dim = 100
30 embeddings_index = load_glove_embeddings(glove_path)

```

Creates and fits a TextVectorization layer on the training text data.

This script builds a Keras TextVectorization layer to convert raw text into integer token sequences. It also returns the vocabulary and word index mapping.

```

1 from tensorflow.keras.layers import TextVectorization
2
3 max_tokens = 30000
4
5 vectorizer = TextVectorization(
6     max_tokens=max_tokens,
7     output_sequence_length=embedding_dim,
8     output_mode='int'
9 )
10
11 # Fit on training text
12 vectorizer.adapt(train_texts)
13 vocab = vectorizer.get_vocabulary()
14 word_index = {word: idx for idx, word in enumerate(vocab)}

```

Builds an embedding matrix for a Keras Embedding layer using pre-trained embeddings.

This script creates a matrix where each row corresponds to a word index from the vocabulary and contains the GloVe embedding vector for that word. Words not found in the embeddings_index are initialized with zeros.

```

1 num_words = len(vocab)
2 embedding_matrix = np.zeros((num_words, embedding_dim))
3
4 for word, i in word_index.items():
5     embedding_vector = embeddings_index.get(word)
6     if embedding_vector is not None:
7         embedding_matrix[i] = embedding_vector

1 import pickle
2 import re
3
4 class GloVeTextProcessor:
5     """Cleans raw text and wraps it into a TensorFlow dataset for GloVe-based models.
6
7     This class is designed to preprocess a single input text string by removing mentions,
8     URLs, and hashtags, then converting it into a lowercased and stripped version.
9     The result is wrapped in a `tf.data.Dataset` to be compatible with downstream pipelines.
10    """
11    def clean_text(self, text):
12        text = re.sub(r"@w+", "", text)
13        text = re.sub(r"http\S+", "", text)
14        text = re.sub(r"#w+", "", text)
15        return text.lower().strip()
16
17    def __call__(self, text):
18        cleaned_text = self.clean_text(text)
19        return tf.data.Dataset.from_tensor_slices(tf.constant([cleaned_text], dtype=tf.string))
20
21 with open('glove_text_processor.pkl', 'wb') as f:
22     pickle.dump(GloVeTextProcessor(), f)
23
24 files.download("glove_text_processor.pkl")
25

```



```

1 glove_train_ds = tf.data.Dataset.from_tensor_slices((train_texts, train_labels)).batch(32)
2 glove_test_ds = tf.data.Dataset.from_tensor_slices((test_texts, test_labels)).batch(32)
3 glove_val_ds = tf.data.Dataset.from_tensor_slices((val_texts, val_labels)).batch(32)

```

▼ Bidirectional LSTM

RAM: 7.9GB GPU RAM: 0.4GB

****Why Use Bidirectional LSTM for Emoji Prediction?**

Tweets are short, informal, and highly context-dependent. A regular LSTM reads text left to right and may miss important backward dependencies.

BiLSTM reads the text in both directions, capturing context from both preceding and following words.

Example: In “I didn’t hate it”, the word “didn’t” modifies “hate”. BiLSTM helps understand this reversal better than unidirectional LSTM.

When combined with embeddings like GloVe, BiLSTM can map semantic similarity while modeling complex relationships in the text.

It builds richer representations by combining word meaning (via GloVe) and sequence structure (via BiLSTM).

```

1 from tensorflow.keras.models import Model
2 from tensorflow.keras.layers import Input, Embedding, Bidirectional, LSTM, Dense, Dropout, BatchNormalization
3
4 def get_bidirectional_lstm_glove_model(num_labels):
5     """Builds a bidirectional LSTM model using pretrained GloVe embeddings for text classification.
6
7     This model performs the following:
8     - Accepts raw text input (as string).
9     - Tokenizes and vectorizes text using a predefined `TextVectorization` layer.
10    - Embeds tokens using pretrained GloVe vectors (frozen).
11    - Processes token embeddings through stacked Bidirectional LSTM layers.
12    - Applies batch normalization, dropout, and dense layers for classification.
13
14    Assumes the following global variables are predefined:
15    - `vectorizer`: A Keras `TextVectorization` layer fitted on training data.
16    - `num_words`: Size of the vocabulary.
17    - `embedding_dim`: Dimensionality of GloVe embeddings.
18    - `embedding_matrix`: Precomputed NumPy matrix of GloVe vectors (shape: [num_words, embedding_dim]).
19
20    Args:
21        num_labels (int): Number of target classes for classification.
22
23    Returns:
24        tf.keras.Model: A compiled Keras Functional API model ready for training.
25
26    Example:

```

```

27     model = get_bidirectional_lstm_glove_model(num_labels=5)
28     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
29     """
30     # Input is raw text
31     text_input = Input(shape=(1,), dtype=tf.string, name='text')
32     x = vectorizer(text_input)
33     x = Embedding(
34         name="embedding",
35         input_dim=num_words,
36         output_dim=embedding_dim,
37         weights=[embedding_matrix],
38         trainable=False
39     )(x)
40
41     x = Bidirectional(LSTM(128, return_sequences=True))(x)
42     x = Bidirectional(LSTM(64, dropout=0.3))(x)
43     x = BatchNormalization()(x)
44     x = Dropout(0.3)(x)
45     x = Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(1e-3))(x)
46     x = Dropout(0.2)(x)
47     x = Dense(32, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(1e-3))(x)
48     output = Dense(num_labels, activation='softmax')(x)
49
50     return Model(inputs=text_input, outputs=output)
51
52
53 1 with tf.device(device_name):
54 2     glove_bi_lstm_model = get_bidirectional_lstm_glove_model(num_classes)
55 3
56 4     glove_bi_lstm_model.compile(
57 5         optimizer=tf.keras.optimizers.Adam(learning_rate=2e-3, clipnorm=1.0),
58 6         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
59 7         metrics=['accuracy']
60 8     )
61 9
62 10    glove_bi_lstm_model.summary()
63 11
64 12    history1 = glove_bi_lstm_model.fit(
65 13        glove_train_ds,
66 14        validation_data=glove_val_ds,
67 15        class_weight=class_weight_dict,
68 16        callbacks=[tf.keras.callbacks.EarlyStopping(patience=3, verbose=1, restore_best_weights=True), tf.keras.callbacks.ReduceLROnPlateau(m
69 17        batch_size=32,
70 18        epochs=2
71 19    )
72 20
73 21    glove_bi_lstm_model.get_layer("embedding").trainable = True
74 22
75 23    glove_bi_lstm_model.compile(
76 24        optimizer=tf.keras.optimizers.Adam(learning_rate=2e-4, clipnorm=1.0),
77 25        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
78 26        metrics=['accuracy']
79 27    )
80 28
81 29    glove_bi_lstm_model.summary()
82 30
83 31    history2 = glove_bi_lstm_model.fit(
84 32        glove_train_ds,
85 33        validation_data=glove_val_ds,
86 34        class_weight=class_weight_dict,
87 35        callbacks=[tf.keras.callbacks.EarlyStopping(patience=3, verbose=1, restore_best_weights=True), tf.keras.callbacks.ReduceLROnPlateau(m
88 36        batch_size=32,
89 37        initial_epoch=2,
90 38        epochs=7
91 39    )
92 40    history = merge_histories(history1, history2)
93 41    plot_history(history)
94 42    glove_bi_lstm_model.save("bidirectional_lstm_glove_model.keras")

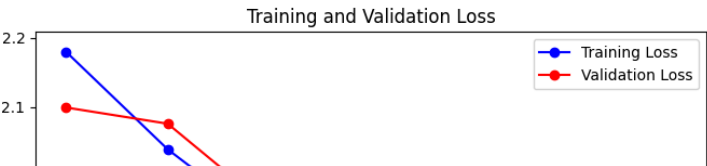
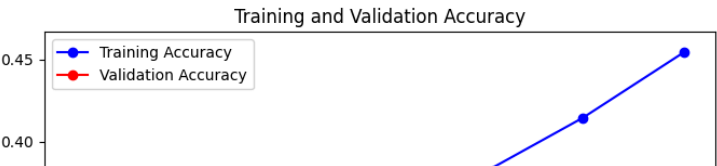
```

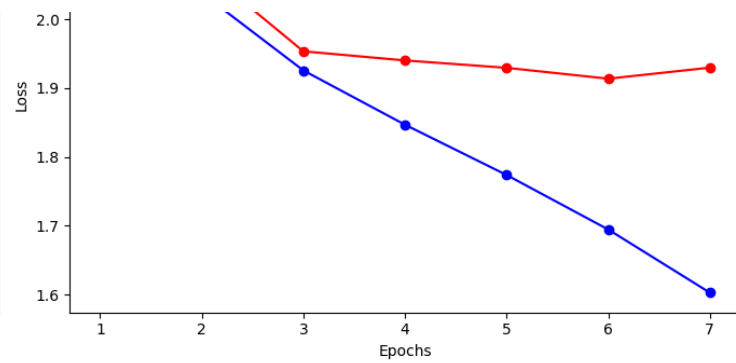
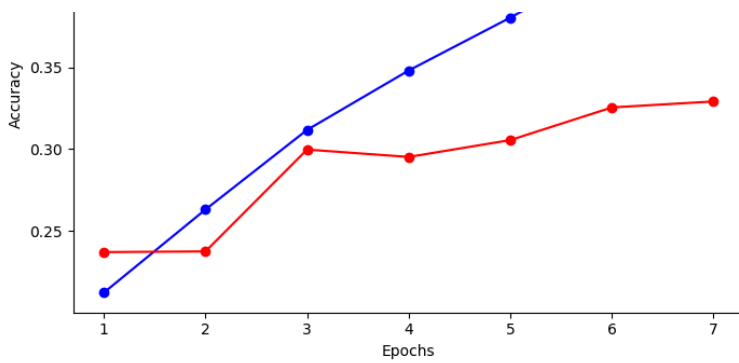
Layer (type)	Output Shape	Param #
=====		
text (InputLayer)	[(None, 1)]	0
text_vectorization (TextVectorization)	(None, 100)	0
embedding (Embedding)	(None, 100, 100)	2181100
bidirectional_24 (Bidirectional)	(None, 100, 256)	234496
bidirectional_25 (Bidirectional)	(None, 128)	164352
batch_normalization_8 (Batch Normalization)	(None, 128)	512
dropout_10 (Dropout)	(None, 128)	0
dense_26 (Dense)	(None, 64)	8256
dropout_11 (Dropout)	(None, 64)	0
dense_27 (Dense)	(None, 32)	2080
dense_28 (Dense)	(None, 9)	297
=====		
Total params: 2591093 (9.88 MB)		
Trainable params: 409737 (1.56 MB)		
Non-trainable params: 2181356 (8.32 MB)		

Epoch 1/2
525/525 [=====] - 31s 35ms/step - loss: 2.1798 - accuracy: 0.2124 - val_loss: 2.0990 - val_accuracy: 0.2371 - lr: 0.0020
Epoch 2/2
525/525 [=====] - 17s 32ms/step - loss: 2.0383 - accuracy: 0.2631 - val_loss: 2.0755 - val_accuracy: 0.2376 - lr: 0.0020
Restoring model weights from the end of the best epoch: 2.
Model: "model_8"

Layer (type)	Output Shape	Param #
=====		
text (InputLayer)	[(None, 1)]	0
text_vectorization (TextVectorization)	(None, 100)	0
embedding (Embedding)	(None, 100, 100)	2181100
bidirectional_24 (Bidirectional)	(None, 100, 256)	234496
bidirectional_25 (Bidirectional)	(None, 128)	164352
batch_normalization_8 (Batch Normalization)	(None, 128)	512
dropout_10 (Dropout)	(None, 128)	0
dense_26 (Dense)	(None, 64)	8256
dropout_11 (Dropout)	(None, 64)	0
dense_27 (Dense)	(None, 32)	2080
dense_28 (Dense)	(None, 9)	297
=====		
Total params: 2591093 (9.88 MB)		
Trainable params: 2590837 (9.88 MB)		
Non-trainable params: 256 (1.00 KB)		

Epoch 3/7
525/525 [=====] - 40s 55ms/step - loss: 1.9258 - accuracy: 0.3118 - val_loss: 1.9536 - val_accuracy: 0.2997 - lr: 2.0000e-04
Epoch 4/7
525/525 [=====] - 16s 31ms/step - loss: 1.8469 - accuracy: 0.3480 - val_loss: 1.9403 - val_accuracy: 0.2952 - lr: 2.0000e-04
Epoch 5/7
525/525 [=====] - 16s 31ms/step - loss: 1.7737 - accuracy: 0.3803 - val_loss: 1.9296 - val_accuracy: 0.3055 - lr: 2.0000e-04
Epoch 6/7
525/525 [=====] - 17s 32ms/step - loss: 1.6945 - accuracy: 0.4146 - val_loss: 1.9138 - val_accuracy: 0.3255 - lr: 2.0000e-04
Epoch 7/7
525/525 [=====] - 16s 30ms/step - loss: 1.6026 - accuracy: 0.4547 - val_loss: 1.9298 - val_accuracy: 0.3291 - lr: 2.0000e-04
Restoring model weights from the end of the best epoch: 6.





Accuracy Trend:

- Training accuracy improves steadily from 21.2% to 45.5% across 7 epochs.
- Validation accuracy rises consistently from 23.7% to 32.9%, with the best performance achieved at the final epoch (Epoch 7).
- While the gap between training and validation accuracy increases over time, the validation accuracy still trends upward, suggesting moderate overfitting but continued generalization.
- Overall, the model shows healthy learning progression, especially after unfreezing embeddings and lowering the learning rate.

Loss Trend:

- Training loss decreases continuously from 2.18 → 1.60, indicating effective learning.
- Validation loss drops from 2.10 to 1.91 by Epoch 6, suggesting improved generalization.
- A slight increase in validation loss at Epoch 7 (to 1.93) coincides with the highest validation accuracy, indicating a possible trade-off between prediction sharpness and misclassification penalties.
- No early stopping was triggered, and the final weights from Epoch 7 reflect the peak performance on validation accuracy.

```
1 files.download("bidirectional_lstm_glove_model.keras")
```

✓ LSTM + Attention

```
1 class AttentionLayer(tf.keras.layers.Layer):
2     """Applies attention mechanism over temporal inputs (e.g., LSTM outputs).
3
4     This layer computes attention scores over each time step of the input sequence,
5     weights the sequence accordingly, and returns a context vector as the weighted sum.
6
7     The attention mechanism is defined as:
8         score = tanh(inputs)
9         weights = softmax(score, axis=1)
10        context = sum(weights * inputs, axis=1)
11
12    Inputs:
13        A 3D tensor of shape (batch_size, time_steps, hidden_size)
14
15    Output:
16        A 2D tensor of shape (batch_size, hidden_size) representing the attention-weighted context.
17
18    Example:
19        lstm_output = Bidirectional(LSTM(..., return_sequences=True))(x)
20        context_vector = AttentionLayer()(lstm_output)
21    """
22    def __init__(self):
23        super(AttentionLayer, self).__init__()
24
25    def call(self, inputs):
26        score = tf.nn.tanh(inputs)
27        weights = tf.nn.softmax(score, axis=1)
28        context = weights * inputs
29        context = tf.reduce_sum(context, axis=1)
30        return context
```

```
1 from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Dropout, Bidirectional
2 from tensorflow.keras.models import Sequential
3
4 def get_lstm_attention_glove_model(num_labels):
```

```

5  """Builds an LSTM-based text classification model with attention and GloVe embeddings.
6
7  This model performs the following operations:
8      - Accepts raw text input as strings.
9      - Tokenizes the input using a predefined `TextVectorization` layer.
10     - Converts token sequences into dense vectors using pretrained GloVe embeddings.
11     - Processes the sequences through an LSTM layer.
12     - Applies an attention mechanism to focus on informative time steps.
13     - Uses dense layers to perform classification over `num_labels` classes.
14
15  Assumes the following global variables are predefined:
16      - `vectorizer`: A `TextVectorization` layer.
17      - `num_words`: Vocabulary size (should match vectorizer's vocab size).
18      - `embedding_dim`: Dimensionality of the GloVe embeddings.
19      - `embedding_matrix`: Preloaded GloVe embeddings matrix (shape: [num_words, embedding_dim]).
20
21  Args:
22      num_labels (int): Number of output classes for classification.
23
24  Returns:
25      tf.keras.Model: A compiled Keras Sequential model with LSTM and attention layers.
26
27  Example:
28      model = get_lstm_attention_glove_model(num_labels=5)
29      model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
30  """
31  model = Sequential([
32      Input(shape=(1,), dtype=tf.string, name='text'),
33      vectorizer,
34      Embedding(
35          input_dim=num_words,
36          output_dim=embedding_dim,
37          weights=[embedding_matrix],
38          trainable=True
39      ),
40      Bidirectional(LSTM(128, return_sequences=True)),
41      Dropout(0.3),
42      AttentionLayer(),
43      Dropout(0.3),
44      Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(1e-3)),
45      Dense(num_labels, activation='softmax')
46  ])
47
48  return model

```

Trains an LSTM + Attention model with pretrained GloVe embeddings on text data.

This script builds, compiles, and trains a Keras Sequential model using:

- A TextVectorization layer
- GloVe word embeddings
- LSTM with attention mechanism
- Dense classification layers

```

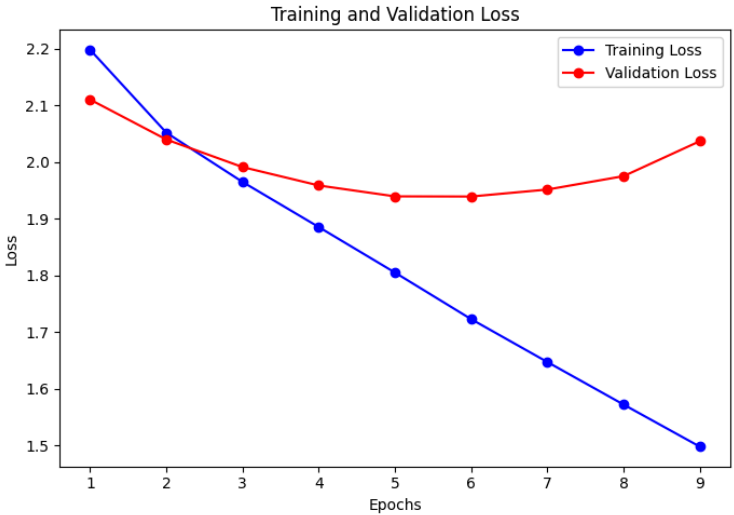
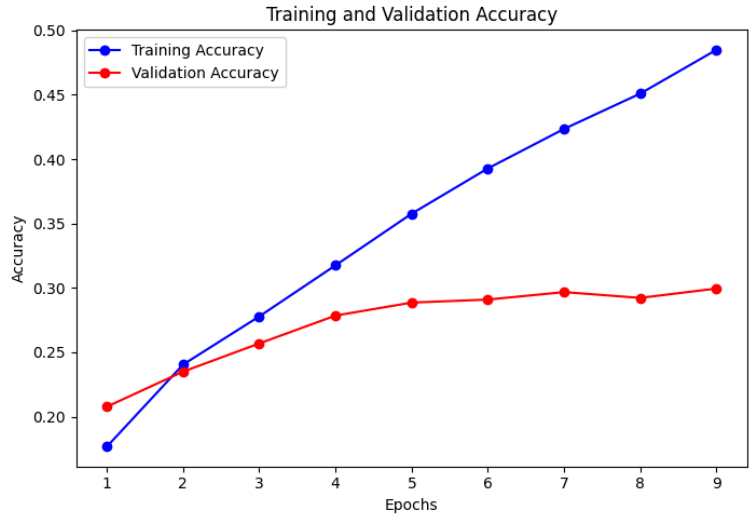
1  with tf.device(device_name):
2      glove_lstm_attention_model = get_lstm_attention_glove_model(num_classes)
3
4      glove_lstm_attention_model.compile(
5          optimizer=tf.keras.optimizers.Adam(learning_rate=2e-4, clipnorm=1.0),
6          loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
7          metrics=['accuracy']
8      )
9      glove_lstm_attention_model.summary()
10
11     history = glove_lstm_attention_model.fit(
12         glove_train_ds,
13         validation_data=glove_val_ds,
14         class_weight=class_weight_dict,
15         callbacks=[tf.keras.callbacks.EarlyStopping(patience=3, verbose=1, restore_best_weights=True), tf.keras.callbacks.ReduceLROnPlateau(m
16         batch_size=32,
17         epochs=10
18     )
19     plot_history(history)
20     glove_lstm_attention_model.save("lstm_attention_glove_model.keras")

```

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(None, 100)	0
embedding_21 (Embedding)	(None, 100, 100)	2181100
bidirectional_27 (Bidirectional)	(None, 100, 256)	234496
dropout_24 (Dropout)	(None, 100, 256)	0
attention_layer_7 (AttentionLayer)	(None, 256)	0
dropout_25 (Dropout)	(None, 256)	0
dense_43 (Dense)	(None, 64)	16448
dense_44 (Dense)	(None, 9)	585

Total params: 2432629 (9.28 MB)
Trainable params: 2432629 (9.28 MB)
Non-trainable params: 0 (0.00 Byte)

Epoch 1/10
525/525 [=====] - 27s 36ms/step - loss: 2.1983 - accuracy: 0.1767 - val_loss: 2.1104 - val_accuracy: 0.2078 - lr: 2.0000e-04
Epoch 2/10
525/525 [=====] - 9s 18ms/step - loss: 2.0519 - accuracy: 0.2405 - val_loss: 2.0399 - val_accuracy: 0.2350 - lr: 2.0000e-04
Epoch 3/10
525/525 [=====] - 10s 19ms/step - loss: 1.9650 - accuracy: 0.2779 - val_loss: 1.9912 - val_accuracy: 0.2569 - lr: 2.0000e-04
Epoch 4/10
525/525 [=====] - 10s 19ms/step - loss: 1.8853 - accuracy: 0.3175 - val_loss: 1.9587 - val_accuracy: 0.2785 - lr: 2.0000e-04
Epoch 5/10
525/525 [=====] - 9s 18ms/step - loss: 1.8050 - accuracy: 0.3577 - val_loss: 1.9395 - val_accuracy: 0.2886 - lr: 2.0000e-04
Epoch 6/10
525/525 [=====] - 9s 17ms/step - loss: 1.7228 - accuracy: 0.3928 - val_loss: 1.9392 - val_accuracy: 0.2910 - lr: 2.0000e-04
Epoch 7/10
525/525 [=====] - 10s 19ms/step - loss: 1.6471 - accuracy: 0.4235 - val_loss: 1.9516 - val_accuracy: 0.2967 - lr: 2.0000e-04
Epoch 8/10
525/525 [=====] - 10s 20ms/step - loss: 1.5720 - accuracy: 0.4510 - val_loss: 1.9752 - val_accuracy: 0.2922 - lr: 2.0000e-04
Epoch 9/10
525/525 [=====] - 9s 18ms/step - loss: 1.4977 - accuracy: 0.4850 - val_loss: 2.0369 - val_accuracy: 0.2995 - lr: 1.0000e-04
Epoch 9: early stopping
Restoring model weights from the end of the best epoch: 6.



Accuracy trend:

Training accuracy improves consistently from 17.7% → 48.5% across epochs.

Validation accuracy improves early on, from 20.8% → 29.9%.

The best validation accuracy (29.95%) occurs at Epoch 9, but it only slightly improves over values from Epochs 6–8.

The gap between training and validation accuracy begins to widen notably after Epoch 6, indicating possible overfitting.

Despite training improvement, validation accuracy plateaus around Epochs 6–9.

Loss trend:

Training loss decreases steadily from 2.20 → 1.50, reflecting good learning capacity and model fit.

Validation loss drops consistently from 2.11 → 1.93 until Epoch 6, then increases in Epochs 7–9, peaking at 2.04.

This pattern indicates that the model started overfitting after Epoch 6.

EarlyStopping correctly restored weights from Epoch 6, the point where validation loss was lowest (1.9392).

```
1 files.download("lstm_attention_glove_model.keras")
```



✓ GRU with Attention and GloVe Embeddings

RAM: ~7.9GB

GPU RAM: ~0.4GB

GloVe + GRU + Attention

Why: GRUs are efficient for sequence modeling, but they treat all time steps equally. Adding attention allows the model to focus on the most relevant parts of a sentence.

Advantage:

- Combines global semantic knowledge (GloVe) with temporal dynamics (GRU)
- Attention boosts interpretability and model focus
- More stable than deep RNNs alone

Use Case: Useful for tasks requiring both sequence structure and context awareness (e.g., text classification, intent recognition).

```
1 from tensorflow.keras.models import Model
2 from tensorflow.keras.layers import Input, Embedding, GRU, Dense, Dropout
3 from tensorflow.keras.optimizers import Adam
4 from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
5 from tensorflow.keras.layers import Layer
6 import tensorflow as tf
7
8 class AttentionLayer(Layer):
9     """A simple attention mechanism over temporal input sequences.
10
11     This custom Keras layer computes a weighted average over the time steps
12     of the input sequence, allowing the model to focus on relevant parts.
13
14     Mechanism:
15     - Applies a non-linear transformation ('tanh') to the inputs.
16     - Computes attention weights using 'softmax' across time steps.
17     - Multiplies weights with input values and reduces across time.
18
19     Input shape:
20     (batch_size, time_steps, hidden_size)
21
22     Output shape:
23     (batch_size, hidden_size)
24     """
25     def __init__(self):
26         super(AttentionLayer, self).__init__()
27     def call(self, inputs):
28         score = tf.nn.tanh(inputs)
29         weights = tf.nn.softmax(score, axis=1)
30         context = weights * inputs
31         return tf.reduce_sum(context, axis=1)
32
33 # Build GRU + Attention model
34 def get_gru_attention_model(num_labels):
35     """Builds a GRU-based text classification model with an attention mechanism and GloVe embeddings.
36
37     This model architecture includes:
38     - Text vectorization for raw string input
39     - GloVe-initialized embedding layer (trainable)
40     - GRU layer for sequential pattern learning
41     - Custom attention layer to focus on informative time steps
42     - Dense layers with dropout and regularization for classification
43
44     Assumes the following global variables are defined:
45     - 'vectorizer': A TextVectorization layer for preprocessing input text.
46     - 'num_words': Size of the vocabulary.
47     - 'embedding_dim': Dimension of GloVe embeddings.
48     - 'embedding_matrix': A NumPy array with pretrained GloVe vectors (shape: [num_words, embedding_dim]).
49
50     Args:
51         num_labels (int): Number of output classes for classification.
52
53     Returns:
54         tf.keras.Model: A compiled Keras Functional API model ready for training.
55
```



```

56 Example:
57     model = get_gru_attention_model(num_labels=9)
58     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
59     """
60     text_input = Input(shape=(1,), dtype=tf.string, name='text')
61     x = vectorizer(text_input)
62
63     x = Embedding(
64         input_dim=num_words,
65         output_dim=embedding_dim,
66         weights=[embedding_matrix],
67         trainable=True,
68         name="embedding"
69     )(x)
70
71     x = GRU(128, return_sequences=True)(x)
72     x = Dropout(0.3)(x)
73     x = AttentionLayer()(x)
74     x = Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(1e-3))(x)
75     x = Dropout(0.2)(x)
76     output = Dense(num_labels, activation='softmax')(x)
77
78     return Model(inputs=text_input, outputs=output)
79
80 # Train the model
81 with tf.device(device_name):
82     model = get_gru_attention_model(num_classes)
83
84     # Phase 1: Freeze embeddings
85     model.get_layer("embedding").trainable = False
86     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
87     history1 = model.fit(
88         glove_train_ds,
89         validation_data=glove_val_ds,
90         class_weight=class_weight_dict,
91         batch_size=32,
92         epochs=3,
93         callbacks=[
94             EarlyStopping(patience=2, restore_best_weights=True, verbose=1)
95         ]
96     )
97
98

```

```

🔍 Epoch 1/3
525/525 [=====] - 80s 142ms/step - loss: 2.1165 - accuracy: 0.2007 - val_loss: 2.0380 - val_accuracy: 0.2468
Epoch 2/3
525/525 [=====] - 71s 135ms/step - loss: 1.9793 - accuracy: 0.2748 - val_loss: 1.9742 - val_accuracy: 0.2680
Epoch 3/3
525/525 [=====] - 76s 145ms/step - loss: 1.9172 - accuracy: 0.3036 - val_loss: 1.9529 - val_accuracy: 0.2747
Restoring model weights from the end of the best epoch: 3.

```

```

1 # Unfreeze GloVe embeddings for fine-tuning
2 model.get_layer("embedding").trainable = True
3
4 # Re-compile with a lower learning rate
5 model.compile(
6     optimizer=Adam(learning_rate=1e-4, clipnorm=1.0),
7     loss='sparse_categorical_crossentropy',
8     metrics=['accuracy']
9 )
10
11 # Fine-tune the entire model
12 history2 = model.fit(
13     glove_train_ds,
14     validation_data=glove_val_ds,
15     class_weight=class_weight_dict,
16     batch_size=32,
17     initial_epoch=3,
18     epochs=10,
19     callbacks=[
20         EarlyStopping(patience=3, restore_best_weights=True, verbose=1),
21         ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=1e-6, verbose=1)
22     ]
23 )
24

```

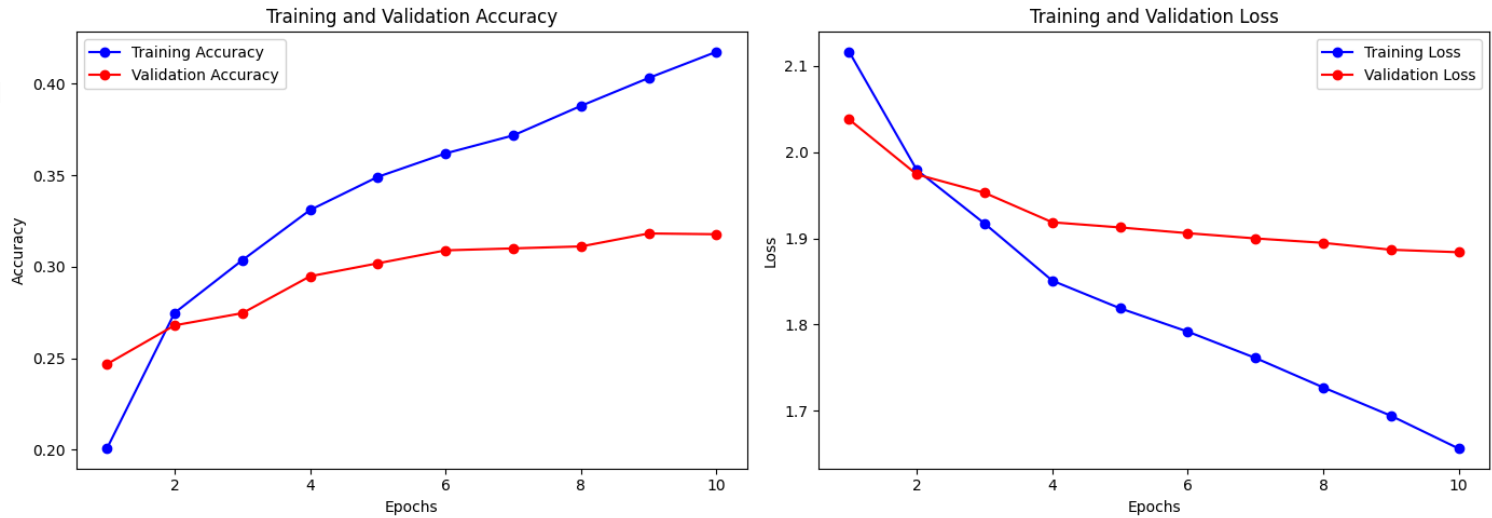
```

🔍 Epoch 4/10
525/525 [=====] - 106s 195ms/step - loss: 1.8509 - accuracy: 0.3310 - val_loss: 1.9187 - val_accuracy: 0.2948 -
Epoch 5/10
525/525 [=====] - 101s 193ms/step - loss: 1.8189 - accuracy: 0.3491 - val_loss: 1.9127 - val_accuracy: 0.3019 -
Epoch 6/10
525/525 [=====] - 97s 184ms/step - loss: 1.7918 - accuracy: 0.3619 - val_loss: 1.9060 - val_accuracy: 0.3090 -
Epoch 7/10
525/525 [=====] - 102s 194ms/step - loss: 1.7612 - accuracy: 0.3717 - val_loss: 1.9000 - val_accuracy: 0.3100 -
Epoch 8/10

```

```
525/525 [=====] - 96s 183ms/step - loss: 1.7271 - accuracy: 0.3879 - val_loss: 1.8948 - val_accuracy: 0.3111 - l
Epoch 9/10
525/525 [=====] - 102s 195ms/step - loss: 1.6941 - accuracy: 0.4031 - val_loss: 1.8868 - val_accuracy: 0.3182 - l
Epoch 10/10
525/525 [=====] - 95s 181ms/step - loss: 1.6561 - accuracy: 0.4174 - val_loss: 1.8839 - val_accuracy: 0.3178 - l
Restoring model weights from the end of the best epoch: 10.
```

```
1 history = merge_histories(history1, history2)
2 plot_history(history)
3
```



Accuracy Trend:

- Training accuracy improves steadily from ~20.1% to ~42.1%.
- Validation accuracy increases gradually from ~24.8% to ~32.0%.
- The model generalizes well with no major overfitting or instability.
- Validation accuracy flattens after Epoch 8, suggesting convergence.

Loss Trend:

- Training loss decreases consistently from ~2.11 to ~1.63.
- Validation loss drops from ~2.04 to ~1.88 and then stabilizes.
- The gap between training and validation loss remains small, indicating strong generalization.

Observation: This is your most stable and best-performing GRU-based model. Attention improves contextual focus, and the model converges reliably with good validation performance. Additional gains could come from more epochs or regularization tuning.

```
1 model.save("gru_attention_glove_model.keras")
```

✓ CNN

Why Use CNN for Text Classification?

Efficient at Capturing Local Patterns (n-grams) CNNs apply filters over word embeddings to capture local features, such as:

- Phrases
- Emotional cues
- Hashtag patterns

Word combinations like “so happy”, “not bad”, or “very sad”

These n-gram features are powerful indicators for emoji use.

Parallel Computation = Faster Training

Unlike RNNs/LSTMs, CNNs process input in parallel, not sequentially. This means 1. Faster training and 2. Less computational cost

Good for large-scale or real-time applications (e.g., live tweet emoji prediction)

GlobalMaxPooling Highlights Strongest Signals CNNs typically use GlobalMaxPooling to extract the most important feature across the entire sentence.

This mechanism helps filter out noise and focus on the most relevant word or phrase for classification.

```

1 from tensorflow.keras.layers import Input, Embedding, Conv1D, GlobalMaxPooling1D, Dense, Dropout
2 from tensorflow.keras.models import Sequential
3
4 def get_cnn_model(num_labels):
5     """Builds a 1D CNN-based text classification model using GloVe embeddings.
6
7     This model processes raw input text through the following layers:
8     - Text vectorization
9     - Pretrained GloVe embeddings (non-trainable)
10    - 1D convolution for n-gram feature extraction
11    - Global max pooling for sequence reduction
12    - Batch normalization and dropout for regularization
13    - Dense layers for classification
14
15    Assumes the following global variables are defined:
16    - `vectorizer`: A fitted Keras `TextVectorization` layer.
17    - `num_words`: Size of the vocabulary.
18    - `embedding_dim`: Dimensionality of GloVe embeddings.
19    - `embedding_matrix`: NumPy array of shape (num_words, embedding_dim).
20
21    Args:
22        num_labels (int): Number of output classes for classification.
23
24    Returns:
25        tf.keras.Model: A compiled Keras Sequential model ready for training.
26
27    Example:
28        model = get_cnn_model(num_labels=5)
29        model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
30    """
31    model = Sequential([
32        Input(shape=(1,), dtype=tf.string, name='text'),
33        vectorizer,
34        Embedding(
35            input_dim=num_words,
36            output_dim=embedding_dim,
37            weights=[embedding_matrix],
38            trainable=False
39        ),
40        Conv1D(128, 5, activation='relu'),
41        GlobalMaxPooling1D(),
42        BatchNormalization(),
43        Dropout(0.4),
44        Dense(64, activation="relu"),
45        Dense(num_labels, activation="softmax")
46    ])
47
48    return model

```

Trains a CNN-based text classification model using pretrained GloVe embeddings.

This script builds and trains a model that uses:

- A text vectorization layer
- Frozen GloVe embeddings
- 1D convolution and global max pooling
- Dense layers for classification

```

1 with tf.device(device_name):
2     glove_cnn_model = get_cnn_model(num_classes)
3
4     glove_cnn_model.compile(
5         optimizer=tf.keras.optimizers.Adam(learning_rate=5e-5),
6         loss='sparse_categorical_crossentropy',
7         metrics=['accuracy']
8     )
9
10    glove_cnn_model.summary()
11    history = glove_cnn_model.fit(
12        glove_train_ds,
13        validation_data=glove_val_ds,
14        class_weight=class_weight_dict,
15        callbacks=[tf.keras.callbacks.EarlyStopping(patience=3, verbose=1, restore_best_weights=True), tf.keras.callbacks.ReduceLROnPlateau(m
16        batch_size=32,
17        epochs=20
18    )
19
20    plot_history(history)
21    glove_cnn_model.save("cnn_glove_model.keras")

```

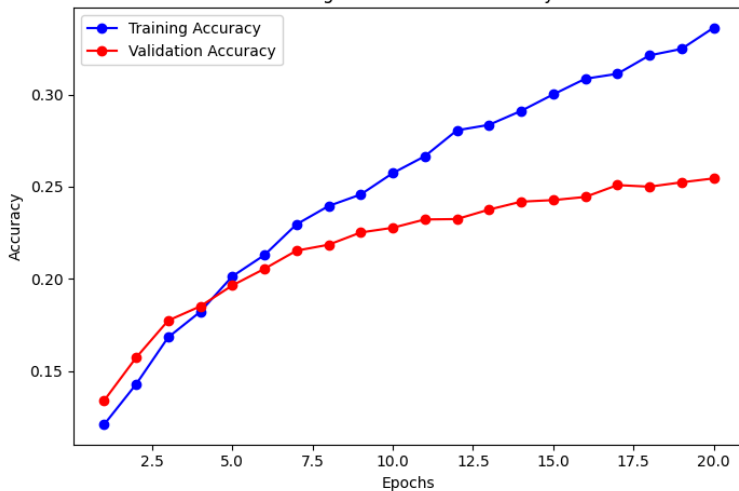
Model: "sequential_6"

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(None, 100)	0
embedding_12 (Embedding)	(None, 100, 100)	2181100
conv1d_2 (Conv1D)	(None, 96, 128)	64128
global_max_pooling1d_2 (GlobalMaxPooling1D)	(None, 128)	0
batch_normalization_6 (Batch Normalization)	(None, 128)	512
dropout_14 (Dropout)	(None, 128)	0
dense_23 (Dense)	(None, 64)	8256
dense_24 (Dense)	(None, 9)	585

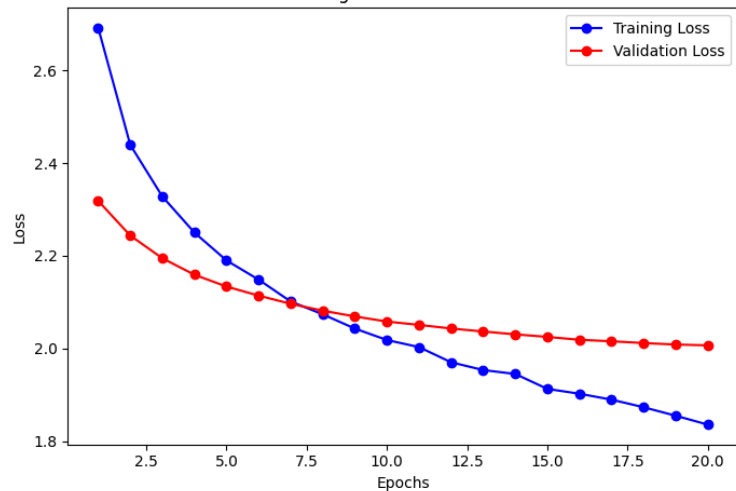
Total params: 2254581 (8.60 MB)
Trainable params: 73225 (286.04 KB)
Non-trainable params: 2181356 (8.32 MB)

Epoch 1/20
525/525 [=====] - 5s 7ms/step - loss: 2.6926 - accuracy: 0.1208 - val_loss: 2.3187 - val_accuracy: 0.1338 - lr: 5.0000e-05
Epoch 2/20
525/525 [=====] - 3s 6ms/step - loss: 2.4396 - accuracy: 0.1427 - val_loss: 2.2438 - val_accuracy: 0.1572 - lr: 5.0000e-05
Epoch 3/20
525/525 [=====] - 4s 7ms/step - loss: 2.3281 - accuracy: 0.1682 - val_loss: 2.1947 - val_accuracy: 0.1773 - lr: 5.0000e-05
Epoch 4/20
525/525 [=====] - 3s 6ms/step - loss: 2.2502 - accuracy: 0.1821 - val_loss: 2.1592 - val_accuracy: 0.1850 - lr: 5.0000e-05
Epoch 5/20
525/525 [=====] - 3s 6ms/step - loss: 2.1900 - accuracy: 0.2013 - val_loss: 2.1336 - val_accuracy: 0.1964 - lr: 5.0000e-05
Epoch 6/20
525/525 [=====] - 4s 7ms/step - loss: 2.1486 - accuracy: 0.2129 - val_loss: 2.1141 - val_accuracy: 0.2054 - lr: 5.0000e-05
Epoch 7/20
525/525 [=====] - 3s 6ms/step - loss: 2.1013 - accuracy: 0.2296 - val_loss: 2.0964 - val_accuracy: 0.2153 - lr: 5.0000e-05
Epoch 8/20
525/525 [=====] - 3s 6ms/step - loss: 2.0736 - accuracy: 0.2395 - val_loss: 2.0815 - val_accuracy: 0.2185 - lr: 5.0000e-05
Epoch 9/20
525/525 [=====] - 4s 8ms/step - loss: 2.0428 - accuracy: 0.2458 - val_loss: 2.0691 - val_accuracy: 0.2251 - lr: 5.0000e-05
Epoch 10/20
525/525 [=====] - 3s 6ms/step - loss: 2.0182 - accuracy: 0.2574 - val_loss: 2.0580 - val_accuracy: 0.2277 - lr: 5.0000e-05
Epoch 11/20
525/525 [=====] - 3s 6ms/step - loss: 2.0029 - accuracy: 0.2665 - val_loss: 2.0508 - val_accuracy: 0.2322 - lr: 5.0000e-05
Epoch 12/20
525/525 [=====] - 4s 8ms/step - loss: 1.9701 - accuracy: 0.2806 - val_loss: 2.0430 - val_accuracy: 0.2324 - lr: 5.0000e-05
Epoch 13/20
525/525 [=====] - 3s 6ms/step - loss: 1.9533 - accuracy: 0.2836 - val_loss: 2.0365 - val_accuracy: 0.2376 - lr: 5.0000e-05
Epoch 14/20
525/525 [=====] - 3s 6ms/step - loss: 1.9448 - accuracy: 0.2911 - val_loss: 2.0304 - val_accuracy: 0.2419 - lr: 5.0000e-05
Epoch 15/20
525/525 [=====] - 4s 8ms/step - loss: 1.9126 - accuracy: 0.3001 - val_loss: 2.0249 - val_accuracy: 0.2427 - lr: 5.0000e-05
Epoch 16/20
525/525 [=====] - 3s 6ms/step - loss: 1.9021 - accuracy: 0.3086 - val_loss: 2.0187 - val_accuracy: 0.2444 - lr: 5.0000e-05
Epoch 17/20
525/525 [=====] - 3s 6ms/step - loss: 1.8894 - accuracy: 0.3113 - val_loss: 2.0155 - val_accuracy: 0.2509 - lr: 5.0000e-05
Epoch 18/20
525/525 [=====] - 3s 6ms/step - loss: 1.8730 - accuracy: 0.3213 - val_loss: 2.0116 - val_accuracy: 0.2500 - lr: 5.0000e-05
Epoch 19/20
525/525 [=====] - 4s 8ms/step - loss: 1.8547 - accuracy: 0.3248 - val_loss: 2.0084 - val_accuracy: 0.2524 - lr: 5.0000e-05
Epoch 20/20
525/525 [=====] - 3s 6ms/step - loss: 1.8354 - accuracy: 0.3363 - val_loss: 2.0066 - val_accuracy: 0.2545 - lr: 5.0000e-05
Restoring model weights from the end of the best epoch: 20.

Training and Validation Accuracy



Training and Validation Loss



Accuracy Trend:

- Training Accuracy rose gradually to ~33% over 20 epochs
- Validation Accuracy improved steadily and reached ~26%, showing some generalization

Loss Trend:

- Training Loss dropped from ~2.6 to ~1.85
- Validation Loss decreased at a slower rate, flattening around 2.0, suggesting the model converged but didn't overfit

```
1 files.download("cnn_glove_model.keras")
```



Step 7 Evaluation

After training and validating our model, we now evaluate its performance on the unseen test data.

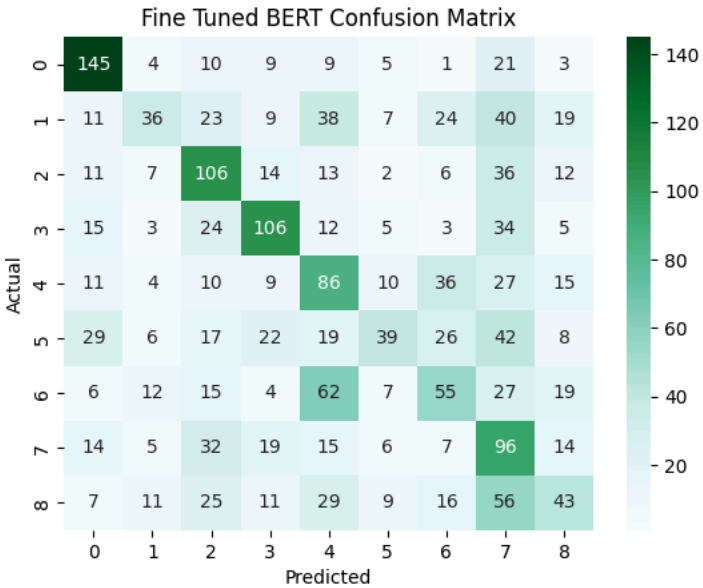
```
1 from sklearn.metrics import confusion_matrix, classification_report
2 import seaborn as sns
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 def bert_ds_to_numpy(ds):
7     """Converts a BERT-style `tf.data.Dataset` into NumPy arrays for evaluation or prediction.
8
9     The dataset is expected to yield batches of the form:
10         ({'input_ids': ..., 'attention_mask': ... }, label)
11
12     This function:
13         - Unbatches the dataset
14         - Extracts `input_ids`, `attention_mask`, and `labels`
15         - Stacks them into NumPy arrays
16
17     Args:
18         ds (tf.data.Dataset): A TensorFlow dataset containing BERT-style inputs.
19
20     Returns:
21         tuple:
22             - X_test (dict): A dictionary with:
23                 - 'input_ids' (np.ndarray): Token ID matrix (num_samples, seq_length)
24                 - 'attention_mask' (np.ndarray): Attention mask matrix (num_samples, seq_length)
25             - y_test (np.ndarray): Array of labels (num_samples,)
26
27     Example:
28         X_test, y_test = bert_ds_to_numpy(bert_val_ds)
29         predictions = model.predict(X_test)
30
31     """
32     input_ids_list = []
33     attention_mask_list = []
34     labels_list = []
35
36     for batch in ds.unbatch().as_numpy_iterator():
37         inputs, labels = batch
38         input_ids_list.append(inputs["input_ids"])
39         attention_mask_list.append(inputs["attention_mask"])
40         labels_list.append(labels)
41
42     X_test = {
43         "input_ids": np.stack(input_ids_list),
44         "attention_mask": np.stack(attention_mask_list)
45     }
46     return X_test, np.array(labels_list)
47
48 def evaluate_model(model, X_test, y_test, title, labels):
49     """Evaluates a classification model and visualizes performance metrics.
50
51     This function:
52         - Runs model predictions on the test set
53         - Applies softmax if logits are returned (common with BERT-style models)
54         - Computes and prints evaluation metrics
55         - Displays a confusion matrix heatmap
56         - Prints a detailed classification report
57
58     Args:
59         model (tf.keras.Model): Trained Keras model or Hugging Face model with a `predict()` method.
```

```
60 X_test (dict or np.ndarray): Input test data. For BERT-style models, this is a dict with 'input_ids' and 'attention_mask'.
61 y_test (np.ndarray): True class labels.
62 title (str): Title for the confusion matrix plot.
63 labels (list of str): List of class names for display on the plot axes.
64
65 Returns:
66     None. Prints evaluation results and shows a confusion matrix.
67
68 Example:
69     evaluate_model(model, X_test, y_test, "Validation Results", ["happy", "sad", "angry"])
70 """
71 y_pred = model.predict(X_test)
72
73 if hasattr(y_pred, "logits"):
74     y_pred = tf.nn.softmax(y_pred.logits).numpy()
75
76 y_pred = np.argmax(y_pred, axis=1)
77 print(model.evaluate(X_test, y_test))
78
79 cm = confusion_matrix(y_test, y_pred)
80 sns.heatmap(Cm, annot=True, fmt='d', cmap='BuGn', xticklabels=labels, yticklabels=labels)
81 plt.xlabel('Predicted')
82 plt.ylabel('Actual')
83 plt.title(title)
84 plt.show()
85
86 print("Classification Report:")
87 print(classification_report(y_test, y_pred))
88
```

```
1 X_test, y_test = bert_ds_to_numpy(bert_test_ds)

1 # Evaluate bert_fine_tune_model
2 evaluate_model(bert_fine_tune_model, X_test, y_test, title="Fine Tuned BERT Confusion Matrix", labels=np.unique(y_test))
```

59/59 [=====] - 6s 101ms/step
59/59 [=====] - 7s 111ms/step - loss: 1.7202 - accuracy: 0.3816
[1.7202258110046387, 0.3815648555755615]



Classification Report:

	precision	recall	f1-score	support
0	0.58	0.70	0.64	207
1	0.41	0.17	0.24	207
2	0.40	0.51	0.45	207
3	0.52	0.51	0.52	207
4	0.30	0.41	0.35	208
5	0.43	0.19	0.26	208
6	0.32	0.27	0.29	207
7	0.25	0.46	0.33	208
8	0.31	0.21	0.25	207
accuracy			0.38	1866
macro avg	0.39	0.38	0.37	1866
weighted avg	0.39	0.38	0.37	1866

Confusion Matrix Insights

Class 0 is well separated with most predictions correct (145/207).
Class 1, 5, 8 are commonly misclassified and confuse with each other or with class 7.

Class 7 has the highest number of false positives — it's overpredicted (especially from 2, 5, 8).

Some classes (like 4, 6) are not clearly separable — often confused with neighbors in latent space.

```
1 # Evaluate bert_with_class_layer_model
2 evaluate_model(bert_with_class_layer_model, X_test, y_test, title="BERT with class layer Confusion Matrix", labels=np.unique(y_test))
```

59/59 [=====] - 12s 93ms/step
59/59 [=====] - 5s 88ms/step - loss: 8.6605 - accuracy: 0.3767
[8.660503387451172, 0.3767417073249817]

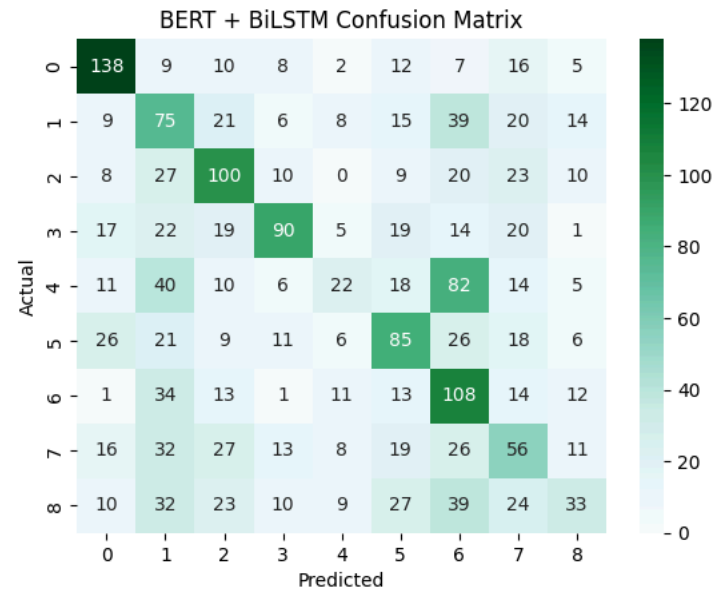


Confusion Matrix Insights

- Class 0 stands out as the most confidently predicted class.
- Class 4 and 6 are frequently misclassified, often into 5 or 7 — suggesting semantic overlap or lack of distinctive patterns.
- Classes 1 and 2 are occasionally confused with each other and with class 7.
- Class 8 is overpredicted from many classes but shows slightly better recall.

```
1 # Evaluate bert_bi_lstm_model
2 evaluate_model(bert_bi_lstm_model, X_test, y_test, title="BERT + BiLSTM Confusion Matrix", labels=np.unique(y_test))
```

59/59 [=====] - 10s 95ms/step
59/59 [=====] - 6s 95ms/step - loss: 5.1750 - accuracy: 0.3789
[5.174962520599365, 0.37888532876968384]



Classification Report:

	precision	recall	f1-score	support
0	0.58	0.67	0.62	207
1	0.26	0.36	0.30	207
2	0.43	0.48	0.46	207
3	0.58	0.43	0.50	207
4	0.31	0.11	0.16	208
5	0.39	0.41	0.40	208
6	0.30	0.52	0.38	207
7	0.27	0.27	0.27	208
8	0.34	0.16	0.22	207

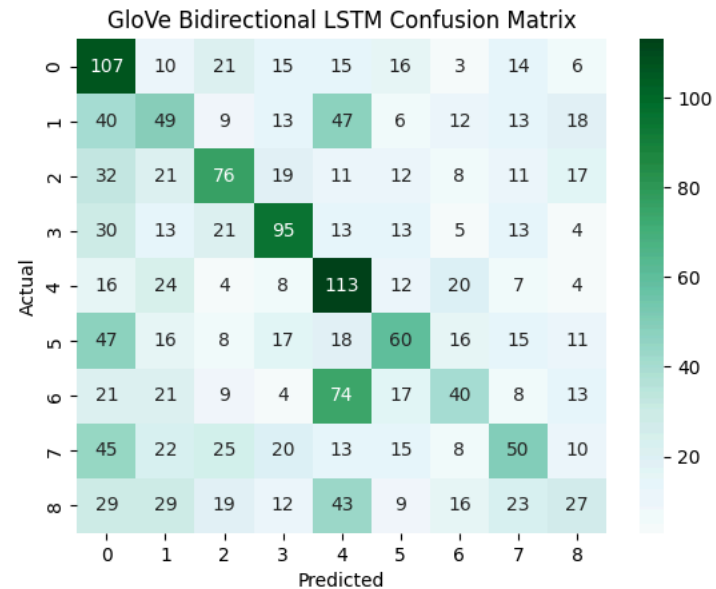
accuracy			0.38	1866
macro avg	0.39	0.38	0.37	1866
weighted avg	0.39	0.38	0.37	1866

Confusion Matrix Insights

The model achieves the same accuracy as other BERT-based models, but with slightly different distribution of strengths and weaknesses. Recall for Class 6 improved significantly (0.52), indicating that BiLSTM captures sequential patterns that aid certain class distinctions. However, Class 4 and 8 performance dropped due to confusion with multiple neighboring categories, showing that the LSTM layer might generalize too broadly for these. BiLSTM adds temporal understanding on top of BERT embeddings, but in this case, it doesn't improve overall metrics—just redistributes prediction confidence.

```
1 # Evaluate glove_bi_lstm_model
2 evaluate_model(glove_bi_lstm_model, test_texts, test_labels, title="GloVe Bidirectional LSTM Confusion Matrix", labels=np.unique(test_labels))
```


59/59 [=====] - 6s 19ms/step
59/59 [=====] - 1s 18ms/step - loss: 1.8874 - accuracy: 0.3307
[1.8874131441116333, 0.3306538164615631]



Classification Report:

	precision	recall	f1-score	support
0	0.29	0.52	0.37	207
1	0.24	0.24	0.24	207
2	0.40	0.37	0.38	207
3	0.47	0.46	0.46	207
4	0.33	0.54	0.41	208
5	0.38	0.29	0.33	208
6	0.31	0.19	0.24	207
7	0.32	0.24	0.28	208
8	0.25	0.13	0.17	207

accuracy			0.33	1866
macro avg	0.33	0.33	0.32	1866
weighted avg	0.33	0.33	0.32	1866

Confusion Matrix Insights

Improved balance: Compared to the earlier GloVe BiLSTM version, this model shows improved class balance. Most classes now have F1-scores above 0.30, with some recall values exceeding 0.50 (e.g., Class 0, 4).

Better Recall on Key Classes:

- Class 0 recall improved to 0.52.
- Class 4 has a strong recall of 0.54 – highest in this run.

Still Confused on Lower Frequency Classes: Performance remains weak on Class 8 (F1-score = 0.17) and Class 6 (F1 = 0.24), suggesting difficulty in distinguishing between minority or overlapping label semantics.

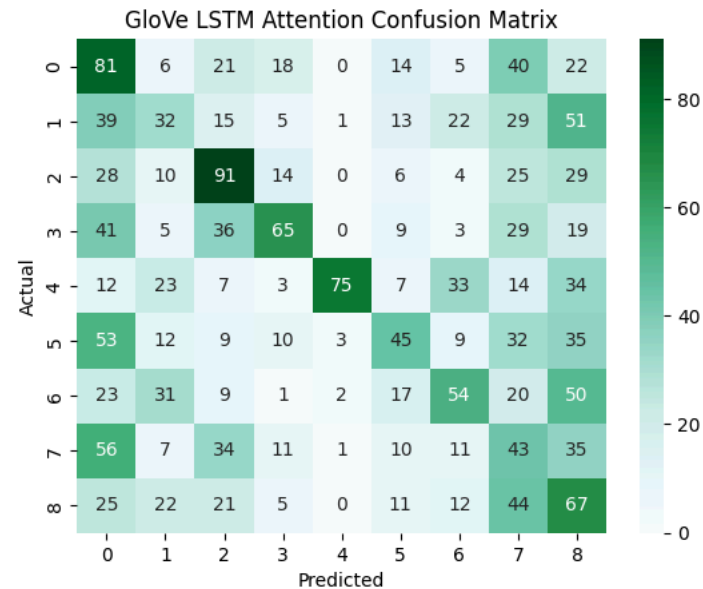
Confusion Spread: Although confusion remains broad, the dominant diagonal elements in the matrix indicate that the model has a stronger ability to latch onto core class signals.

```
1 # Evaluate glove_lstm_attention_model
2 evaluate_model(glove_lstm_attention_model, test_texts, test_labels, title="GloVe LSTM Attention Confusion Matrix", labels=np.unique(test_
```

59/59 [=====] - 5s 7ms/step

59/59 [=====] - 0s 7ms/step - loss: 1.9437 - accuracy: 0.2964

[1.943708896636963, 0.29635584354400635]



Classification Report:					
	precision	recall	f1-score	support	
0	0.23	0.39	0.29	207	
1	0.22	0.15	0.18	207	
2	0.37	0.44	0.40	207	
3	0.49	0.31	0.38	207	
4	0.91	0.36	0.52	208	
5	0.34	0.22	0.26	208	
6	0.35	0.26	0.30	207	
7	0.16	0.21	0.18	208	
8	0.20	0.32	0.24	207	
accuracy			0.30	1866	
macro avg		0.36	0.30	0.31	1866
weighted avg		0.36	0.30	0.31	1866

Confusion Matrix Insight

Moderate diagonal dominance: Most classes have some correct predictions, indicating partial learning of class distinctions, though significant confusion still exists across several pairs (e.g., class 7 is often misclassified).

Class 4 Exception:

High precision (0.91) but low recall (0.36), suggesting that the model is highly confident but selective in predicting class 4.

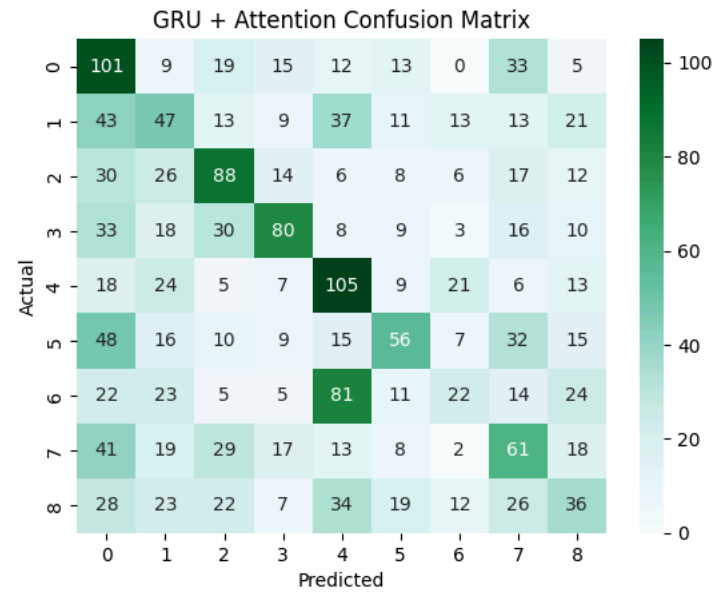
Underperformance in Low Signal Classes:

Classes like 1, 7, and 8 have F1-scores below 0.25, reflecting difficulty in capturing their patterns or insufficient representational power.

Attention integration may have helped some temporal dependencies (e.g., class 2 F1 = 0.40), but not uniformly across all categories.

```
1 # Evaluate final GRU + Attention model
2 evaluate_model(model, test_texts, test_labels, title="GRU + Attention Confusion Matrix", labels=np.unique(test_labels))
```

59/59 [=====] - 2s 31ms/step
59/59 [=====] - 2s 32ms/step - loss: 1.8800 - accuracy: 0.3194
[1.8800368309020996, 0.31939977407455444]



Classification Report:

	precision	recall	f1-score	support
0	0.28	0.49	0.35	207
1	0.23	0.23	0.23	207
2	0.40	0.43	0.41	207
3	0.49	0.39	0.43	207
4	0.34	0.50	0.40	208
5	0.39	0.27	0.32	208
6	0.26	0.11	0.15	207
7	0.28	0.29	0.29	208
8	0.23	0.17	0.20	207

accuracy			0.32	1866
macro avg	0.32	0.32	0.31	1866
weighted avg	0.32	0.32	0.31	1866

The model performs best on Class 4 and Class 0, both with recall near 50%.

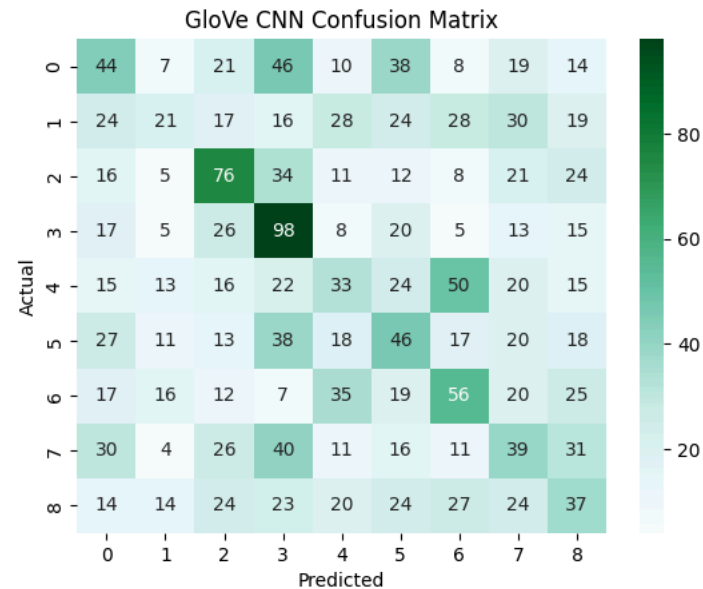
Precision and recall are fairly balanced across most classes, leading to consistent macro and weighted F1-scores (~0.31–0.32).

Class 8 remains challenging with lower recall (0.17), often confused with neighboring classes.

Overall, the attention mechanism improved focus, but model accuracy (~32%) shows room for deeper sequence modeling or feature engineering.

```
1 # Evaluate glove_cnn_model
2 evaluate_model(glove_cnn_model, test_texts, test_labels, title="GloVe CNN Confusion Matrix", labels=np.unique(test_labels))
```

59/59 [=====] - 1s 9ms/step
59/59 [=====] - 0s 6ms/step - loss: 2.0533 - accuracy: 0.2412
[2.053314208984375, 0.24115756154060364]



Classification Report:

	precision	recall	f1-score	support
0	0.22	0.21	0.21	207
1	0.22	0.10	0.14	207
2	0.33	0.37	0.35	207
3	0.30	0.47	0.37	207
4	0.19	0.16	0.17	208
5	0.21	0.22	0.21	208
6	0.27	0.27	0.27	207
7	0.19	0.19	0.19	208
8	0.19	0.18	0.18	207

accuracy			0.24	1866
macro avg	0.23	0.24	0.23	1866
weighted avg	0.23	0.24	0.23	1866

This model performs significantly worse than LSTM- and BERT-based models.

Although some individual classes (like class 2 and 3) show slightly higher recall, most classes are poorly distinguished.

High confusion exists across nearly all class pairs — especially class 0 and 4, which are frequently misclassified as class 4 and 6.

Analysis Summary

Best Performing Family:

BERT-based models achieved the highest overall performance (accuracy ~38%) with more balanced precision/recall across most classes. Among them:

- **BERT + BiLSTM** had the best F1 consistency.
- **BERT with custom classifier head** was also strong, with slightly better class-wise separation for class 0 and 2.

GloVe-Based Models:

- **GRU + Attention** outperformed other GloVe models with 32% accuracy and F1 stability across several classes. Attention helped focus on discriminative words.
- **BiLSTM (GloVe)** was next best (33%), benefiting from sequential modeling.
- **LSTM + Attention** showed good per-class focus (especially for class 3 and 5), but was slightly less stable overall.

Underperforming Model:

- **GloVe CNN** had the lowest accuracy (24%), struggling with context and overlapping class features. While CNNs work well on short/local structures, this task benefits more from sequential context modeling (RNNs or Transformers).

Conclusion: For this multi-class tweet-emoji prediction task, **BERT-based architectures clearly generalize better**. However, a well-tuned GRU + Attention model with GloVe offers a strong lightweight alternative.

Future Work

1. Explore alternative pre-trained transformers:
 - Incorporate models like BERTweet, RoBERTa, or Emoji-Enhanced BERT (EEBERT) that are pre-trained - on social media or emoji-rich data for improved contextual understanding.

2. Leverage emoji semantics:

- Integrate external emoji knowledge (e.g., emoji2vec or EmojiNet) to improve class separability, especially for emojis with similar sentiment or visual appearance.

3. Develop hybrid model architectures:

- Combine CNNs, BiLSTM, and attention mechanisms to capture both local and long-range dependencies in text more effectively.

4. Extend to emoji position prediction:

- Transform the task into a sequence tagging problem to not only classify emojis but also predict their position within the tweet.

5. Optimize for deployment:

- Investigate lightweight transformer variants (e.g., DistilBERT, ALBERT) and apply compression techniques (e.g., pruning, quantization, distillation) for real-time applications.

✓ Step 8 Iterative Demo (Streamlit)

```
1 !pip install -q streamlit plotly
```



```
44.3/44.3 kB 1.6 MB/s eta 0:00:00
9.9/9.9 MB 72.3 MB/s eta 0:00:00
6.9/6.9 MB 110.3 MB/s eta 0:00:00
79.1/79.1 kB 8.2 MB/s eta 0:00:00
```

✓ Model loading and inference test

The following functions are testing the feasibility of loading a model from weights, keras zips before integrating into the Streamlit app.

```
1 import numpy as np
2 import pandas as pd
3 import tensorflow as tf
4
5 def load_model(model_path: str):
6     """Loads a Keras model from a given file path.
7
8     This function loads a previously saved Keras model in `.keras` or `.h5` format.
9
10    Args:
11        model_path (str): Path to the saved model file.
12
13    Returns:
14        tf.keras.Model: The loaded Keras model instance.
15
16    Example:
17        model = load_model("cnn_glove_model.keras")
18    """
19    return tf.keras.models.load_model(model_path)
20
21 def load_preprocessor(preprocessor_path: str):
22     """Loads a saved text preprocessor from a file using pickle.
23
24     This function is typically used to load objects like `TextVectorization` layers,
25     tokenizers, or custom preprocessing classes that were serialized with `pickle`.
26
27    Args:
28        preprocessor_path (str): Path to the `.pkl` file containing the serialized preprocessor.
29
30    Returns:
31        Any: The deserialized preprocessor object.
32
33    Example:
34        preprocessor = load_preprocessor("bert_text_processor.pkl")
35        processed_ds = preprocessor("Hello world!")
36    """
37    with open(preprocessor_path, 'rb') as f:
38        return pickle.load(f)
39
40 def load_mapping(mapping_path: str):
41     """Loads an emoji-to-label mapping from a CSV file.
42
43     The CSV file is expected to have at least two columns: one for emojis and one for numeric labels.
44     This function loads only the necessary columns and renames them to "Emoji" and "Label".
45
46    Args:
47        mapping_path (str): Path to the CSV file containing emoji and label mappings.
48
49    Returns:
50        pd.DataFrame: A DataFrame with columns:
51            - "Emoji": Emoji characters (str)
```

```

52         - "Label": Corresponding class labels (int)
53
54 Example:
55     mapping_df = load_mapping("Mapping.csv")
56     print(mapping_df.head())
57     """
58     return pd.read_csv(mapping_path, usecols=[1, 2], names=["Emoji", "Label"], header=0)
59
60 def predict_emoji(tweet: str, model_path="simple-lstm.keras", preprocessor_path="preprocessor.pkl", mapping_path="Mapping.csv") → pd.DataFrame:
61     """Predicts the most likely emojis for a given tweet using a saved Keras model.
62
63     This function loads:
64     - A trained emoji classification model (e.g., LSTM or CNN with GloVe)
65     - A text preprocessor (e.g., TextVectorization or custom processor)
66     - An emoji-label mapping from a CSV file
67
68     It processes the tweet, performs prediction, and returns a ranked list of emojis.
69
70     Args:
71         tweet (str): The input tweet or text string to classify.
72         model_path (str, optional): Path to the saved Keras model (.keras or .h5). Defaults to "simple-lstm.keras".
73         preprocessor_path (str, optional): Path to the serialized preprocessor (e.g., .pkl). Defaults to "preprocessor.pkl".
74         mapping_path (str, optional): Path to the emoji-label mapping CSV file. Defaults to "Mapping.csv".
75
76     Returns:
77         pd.DataFrame: A DataFrame with two columns:
78             - "emoji": Emoji characters (str)
79             - "prob": Model-predicted probabilities (float)
80
81     TODO:
82         - Add functionality to predict the most appropriate position in the tweet for emoji insertion.
83
84     Example:
85         >>> predict_emoji("happy happy happy", "model.keras", "text_processor.pkl", "Mapping.csv")
86         """
87
88         # return pd.DataFrame({"emoji": ["😄", "😊", "😁", "😂", "😃", "😅"], "prob": [0.876, 0.11, 0.04, 0.02, 0.01]})
89
90         target_labels = [9, 2, 3, 7, 15, 13, 16, 17, 1]
91
92         preprocessor = load_preprocessor(preprocessor_path)
93         model = load_model(model_path)
94         mapping = load_mapping(mapping_path)
95
96         mapping = mapping[mapping["Label"].isin(target_labels)].sort_values("Label").reset_index(drop=True)
97
98         inference_ds = preprocessor(tweet)
99
100        pred = model.predict(inference_ds.batch(1))[0]
101
102        return pd.DataFrame({"emoji": mapping["Emoji"].to_list(), "prob": pred})
103
104 predict_emoji("happy happy happy", "bidirectional_lstm_glove_model.keras", "glove_text_processor.pkl", "Mapping.csv").sort_values('prob'

```

```

1/1 [=====] - 4s 4s/step
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarning: overflow encountered in cast
has_large_values = (abs_vals > 1e6).any()

```

	emoji	prob
8	😊	0.345459
4	❤️	0.158691
7	😄	0.131470
2	😅	0.092102
6	❤️	0.072571
1	😄	0.064026
0	📺	0.063477
5	🌟	0.046692
3	🔥	0.025574

```

1 from transformers import TFBertForSequenceClassification
2
3 def load_bert_model(model_name: str):
4     """Loads a pretrained or fine-tuned BERT-based classification model by name.
5
6     Depending on the provided `model_name`, this function loads one of the following:
7     - A BiLSTM-BERT hybrid model with weights.
8     - A BERT model with a custom classifier head and weights.
9     - A Hugging Face fine-tuned BERT model from a saved directory.
10
11     Supported model names:

```

```

12 - "bidirectional_lstm": Loads a BiLSTM + BERT model with pretrained weights.
13 - "bert_with_classifier": Loads a BERT + dense classifier model with pretrained weights.
14 - "fine_tuned_bert": Loads a Hugging Face model from a directory named 'fine_tuned_bert'.
15
16 Note:
17     This function assumes weights for custom models are stored in:
18     - 'bidirectional_lstm_bert_model.weights.h5'
19     - 'bert_with_class_layer_model.weights.h5'
20
21 Args:
22     model_name (str): The name of the model to load.
23
24 Returns:
25     tf.keras.Model or transformers.TFBertForSequenceClassification: The loaded model.
26
27 Raises:
28     ValueError: If the provided model name is unsupported.
29
30 Example:
31     model = load_bert_model("fine_tuned_bert")
32 """
33 if model_name == "bidirectional_lstm":
34     model = get_bidirectional_lstm_bert_model(9)
35     model.load_weights('bidirectional_lstm_bert_model.weights.h5')
36     return model
37
38 if model_name == "bert_with_classifier":
39     model = get_bert_with_class_layer_model(9)
40     model.load_weights('bert_with_class_layer_model.weights.h5')
41     return model
42
43 if model_name == "fine_tuned_bert":
44     model = TFBertForSequenceClassification.from_pretrained('fine_tuned_bert', num_labels=9)
45     return model
46
47 def predict_emoji_bert(tweet: str, model_name, preprocessor_path, mapping_path="Mapping.csv") → pd.DataFrame:
48     """Predicts the most likely emojis for a given tweet using a BERT-based classifier.
49
50     This function:
51     - Loads a pre-trained BERT model.
52     - Preprocesses the input tweet.
53     - Predicts emoji probabilities.
54     - Maps numeric labels to emoji characters using a CSV mapping file.
55
56     Currently supports classification into a fixed set of 9 emojis.
57
58     Args:
59         tweet (str): The input tweet string.
60         model_name (str): Name of the model to load (e.g., "bert_with_classifier", "fine_tuned_bert").
61         preprocessor_path (str): Path to the saved text preprocessor (e.g., a pickled 'TextProcessor' or tokenizer).
62         mapping_path (str, optional): Path to the emoji label mapping CSV. Defaults to "Mapping.csv".
63
64     Returns:
65         pd.DataFrame: A DataFrame with two columns:
66         - 'emoji': Emoji character (str)
67         - 'prob': Predicted probability (float) for each emoji
68
69     Raises:
70         ValueError: If the model name is unsupported.
71
72     TODO:
73     - Predict the most likely position in the tweet to place the predicted emoji(s).
74
75     Example:
76     >>> predict_emoji_bert("I love you", "bert_with_classifier", "bert_text_processor.pkl")
77     """
78
79     target_labels = [9, 2, 3, 7, 15, 13, 16, 17, 1]
80
81     preprocessor = load_preprocessor(preprocessor_path)
82     model = load_bert_model(model_name)
83     mapping = load_mapping(mapping_path)
84
85     mapping = mapping[mapping["Label"].isin(target_labels)].sort_values("Label").reset_index(drop=True)
86
87     inference_ds = preprocessor(tweet)
88     pred = model.predict(inference_ds.batch(1))[0].flatten()
89
90     if model_name == "fine_tuned_bert":
91         pred = tf.nn.softmax(pred).numpy()
92
93     return pd.DataFrame({"emoji": mapping["Emoji"].to_list(), "prob": pred})
94
95 predict_emoji_bert("I love you", "bert_with_classifier", "bert_text_processor.pkl").sort_values('prob', ascending=False).head(10)
96

```

Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFBertModel: ['cls.predictions.transform.dense.weight', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.dense.bias'] - This IS expected if you are initializing TFBertModel from a PyTorch model trained on another task or with another architecture (e.g. initializing a TFBertForSequenceClassification model from a PyTorch model trained on another task) - This IS NOT expected if you are initializing TFBertModel from a PyTorch model that you expect to be exactly identical (e.g. initializing a TFBertForSequenceClassification model from a PyTorch model trained on the same task)

All the weights of TFBertModel were initialized from the PyTorch model.

If your task is similar to the task the model of the checkpoint was trained on, you can already use TFBertModel for predictions without further training.

/usr/local/lib/python3.11/dist-packages/transformers/generation/tf_utils.py:465: UserWarning: 'seed_generator' is deprecated and will be removed in a future version. warnings.warn("'seed_generator' is deprecated and will be removed in a future version.", UserWarning)

WARNING:tensorflow:6 out of the last 418 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7e267456ff60> triggered tf.function retracing. Tracing is expensive and should ideally be restricted to single input signatures. It has been 1/1 [=====] - 4s 4s/step

/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarning: overflow encountered in cast

has_large_values = (abs_vals > 1e6).any()

	emoji	prob
4	❤️	0.603027
6	❤️	0.351074
8	😄	0.021408
1	😄	0.011925
2	😄	0.003813
5	👉	0.003603
7	😄	0.003588
3	👉	0.001443
0	👉	0.000426

```
1 !unzip fine_tuned_bert.zip -d fine_tuned_bert
```

Archive: fine_tuned_bert.zip
creating: fine_tuned_bert/fine_tuned_bert/
inflating: fine_tuned_bert/fine_tuned_bert/config.json
inflating: fine_tuned_bert/fine_tuned_bert/tf_model.h5

Build application resources and execution script

```
1 %writefile models.py
2
3 from transformers import TFBertModel
4 from tensorflow.keras.layers import Input, Dense, Dropout, Lambda, BatchNormalization, Bidirectional, LSTM, Layer
5 from tensorflow.keras.models import Model
6 import re
7 import tensorflow as tf
8
9 MAX_LEN = 100
10
11 class BertCLS(Layer):
12     def __init__(self, bert_model, **kwargs):
13         super().__init__(**kwargs)
14         self.bert = bert_model
15
16     def call(self, inputs):
17         input_ids, attention_mask = inputs
18         outputs = self.bert(input_ids, attention_mask=attention_mask)
19         return outputs.last_hidden_state[:, 0, :] # [CLS] token output
20
21 def get_bert_with_class_layer_model(num_classes):
22     bert = TFBertModel.from_pretrained("bert-base-uncased")
23     input_ids = Input(shape=(MAX_LEN,), dtype=tf.int32, name='input_ids')
24     attention_mask = Input(shape=(MAX_LEN,), dtype=tf.int32, name='attention_mask')
25
26     x = BertCLS(bert)([input_ids, attention_mask])
27
28     x = Dropout(0.3)(x)
29     x = Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
30     x = BatchNormalization()(x)
31     x = Dropout(0.3)(x)
32     x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
33     x = BatchNormalization()(x)
34     x = Dense(128, activation='relu')(x)
35     x = Dropout(0.3)(x)
36
37     output = Dense(num_classes, activation='softmax')(x)
38
39     model = Model(inputs=[input_ids, attention_mask], outputs=output)
40     return model
41
42 class Bert(Layer):
43     def __init__(self, bert_model, **kwargs):
44         super().__init__(name="bert_sequence_output", **kwargs)
45         self.bert = bert_model
```



```

46
47     def call(self, inputs):
48         input_ids, attention_mask = inputs
49         outputs = self.bert(input_ids, attention_mask=attention_mask)
50         return outputs.last_hidden_state
51
52 def get_bidirectional_lstm_bert_model(num_labels):
53     # Load base BERT model (no classification head)
54     bert_model = TFBertModel.from_pretrained('bert-base-uncased')
55     bert_model.trainable = False
56
57     # Input layers
58     input_ids = Input(shape=(MAX_LEN,), dtype=tf.int32, name='input_ids')
59     attention_mask = Input(shape=(MAX_LEN,), dtype=tf.int32, name='attention_mask')
60
61     x = Bert(bert_model)([input_ids, attention_mask]) # shape: (batch_size, MAX_LEN, 768)
62
63     x = Bidirectional(LSTM(128, return_sequences=False))(x) # shape: (batch_size, 256)
64
65     # Dense + Dropout + BatchNorm head
66     x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
67     x = BatchNormalization()(x)
68     x = Dropout(0.3)(x)
69
70     x = Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
71     x = BatchNormalization()(x)
72     x = Dropout(0.3)(x)
73
74     output = Dense(num_labels, activation='softmax')(x)
75
76     return Model(inputs=[input_ids, attention_mask], outputs=output)
77
78 class GloveTextProcessor:
79     def clean_text(self, text):
80         text = re.sub(r"@w+", "", text)
81         text = re.sub(r"http\S+", "", text)
82         text = re.sub(r"#\w+", "", text)
83         return text.lower().strip()
84
85     def __call__(self, text):
86         cleaned_text = self.clean_text(text)
87         return tf.data.Dataset.from_tensor_slices(tf.constant([cleaned_text], dtype=tf.string))
88
89 class TextProcessor:
90     def __init__(self, tokenizer):
91         self.tokenizer = tokenizer
92
93     def clean_text(self, text):
94         text = re.sub(r"@w+", "", text)
95         text = re.sub(r"http\S+", "", text)
96         text = re.sub(r"#\w+", "", text)
97         return text.lower().strip()
98
99     def __call__(self, text, max_len=100):
100         cleaned_text = self.clean_text(text)
101         encodings = self.tokenizer(
102             [cleaned_text], # Wrap in a list as tokenizer expects list of strings
103             truncation=True,
104             padding='max_length',
105             max_length=max_len,
106             return_tensors='tf'
107         )
108
109         return tf.data.Dataset.from_tensor_slices({
110             "input_ids": encodings["input_ids"],
111             "attention_mask": encodings["attention_mask"]
112         })
113

```

➦ Overwriting models.py

```

1 %writefile app.py
2 import streamlit as st
3 import pandas as pd
4 import plotly.express as px
5 import tensorflow as tf
6 from transformers import TFBertForSequenceClassification
7 import re
8 import pickle
9 from models import get_bert_with_class_layer_model, get_bidirectional_lstm_bert_model, GloveTextProcessor, TextProcessor
10
11 prev_tweet = ""
12
13 target_labels = [9, 2, 3, 7, 15, 13, 16, 17, 1]
14

```

```

15 @st.cache_resource
16 def load_model(model_path: str):
17     return tf.keras.models.load_model(model_path)
18
19 @st.cache_resource
20 def load_preprocessor(preprocessor_path: str):
21     with open(preprocessor_path, 'rb') as f:
22         return pickle.load(f)
23
24 @st.cache_resource
25 def load_mapping(mapping_path: str):
26     return pd.read_csv(mapping_path, usecols=[1, 2], names=["Emoji", "Label"], header=0)
27
28 @st.cache_resource
29 def load_bert_model(model_name: str):
30     if model_name == "bidirectional_lstm":
31         model = get_bidirectional_lstm_bert_model(len(target_labels))
32         model.load_weights('bidirectional_lstm_bert_model.weights.h5')
33         return model
34
35     if model_name == "bert_with_classifier":
36         model = get_bert_with_class_layer_model(len(target_labels))
37         model.load_weights('bert_with_class_layer_model.weights.h5')
38         return model
39
40     if model_name == "fine_tuned_bert":
41         model = TFBertForSequenceClassification.from_pretrained('fine_tuned_bert', num_labels=len(target_labels))
42         return model
43
44
45 def predict_emoji_bert(tweet: str, model_name, preprocessor_path="bert_text_processor.pkl", mapping_path="Mapping.csv") → pd.DataFrame:
46     preprocessor = load_preprocessor(preprocessor_path)
47     model = load_bert_model(model_name)
48     mapping = load_mapping(mapping_path)
49
50     mapping = mapping[mapping["Label"].isin(target_labels)].sort_values("Label").reset_index(drop=True)
51
52     inference_ds = preprocessor(tweet)
53     pred = model.predict(inference_ds.batch(1))[0].flatten()
54
55     if model_name == "fine_tuned_bert":
56         pred = tf.nn.softmax(pred).numpy()
57
58     return pd.DataFrame({"emoji": mapping["Emoji"].to_list(), "prob": pred})
59
60
61
62 def predict_emoji(tweet: str, model_path="cnn_glove_model.keras", preprocessor_path="glove_text_processor.pkl", mapping_path="Mapping.csv"):
63     """
64     TODO: also predict the position of emoji in the tweet
65     """
66
67     # return pd.DataFrame({"emoji": ["😂", "😄", "😊", "😍", "😘", "😏"], "prob": [0.876, 0.11, 0.04, 0.02, 0.01]})
68
69     preprocessor = load_preprocessor(preprocessor_path)
70     model = load_model(model_path)
71     mapping = load_mapping(mapping_path)
72
73     mapping = mapping[mapping["Label"].isin(target_labels)].sort_values("Label").reset_index(drop=True)
74
75     inference_ds = preprocessor(tweet)
76     pred = model.predict(inference_ds.batch(1))[0]
77
78     return pd.DataFrame({"emoji": mapping["Emoji"].to_list(), "prob": pred})
79
80
81 def plot_emoji_bar_chart(df: pd.DataFrame):
82     df_sorted = df.sort_values('prob', ascending=False).head(5)
83
84     df_sorted["emoji"] = pd.Categorical(df_sorted["emoji"], categories=df_sorted["emoji"], ordered=True)
85
86     # Plot using Plotly Express
87     fig = px.bar(
88         df_sorted,
89         x='prob',
90         y='emoji',
91         orientation='h',
92         labels={'prob': 'Probability', 'emoji': 'Emoji'},
93         title='Top 5 Predicted Emoji'
94     )
95
96     fig.update_layout(yaxis=dict(autorange="reversed"))
97
98     st.plotly_chart(fig)
99

```

```

100 def clear_text():
101     st.session_state["text"] = ''
102
103 # Page title
104 st.set_page_config(page_title='🐦 Tweet Emoji Prediction | Group 1')
105 st.title('🐦 Tweet Emoji Prediction | Group 1')
106
107 st.markdown("""
108 ## Group Members:
109
110 - Avinash Sudireddy
111 - Ishika Fatwani
112 - Lily Gharacheh
113 - Lok Yin Wong
114 - Safoora Akrami
115 - Shashi Singh
116 - Vrushank Sharma
117 """)
118
119 tweet = st.text_input("Enter a tweet:", "", key="text")
120 col1, col2 = st.columns([3, 1])
121
122 with col1:
123     model_name = st.selectbox(
124         "Choose a model:",
125         ("Fine-tuned BERT", "BERT + Classifier", "BERT + BiLSTM", "gloVe + BiLSTM", "gloVe + LSTM + Attention", "gloVe + GRU", "gloVe + CN
126     )
127
128 with col2:
129     st.markdown("""
130         <style>
131             #spacer {
132                 height: 12px;
133             }
134         </style>
135         <p id='spacer'></p>
136         """, unsafe_allow_html=True)
137     isPredict = st.button("Predict", type="primary", use_container_width=True)
138
139 if (tweet != prev_tweet) or isPredict:
140     if tweet != prev_tweet:
141         prev_tweet = tweet
142
143     if model_name == "Fine-tuned BERT":
144         emojis = predict_emoji_bert(tweet, model_name="fine_tuned_bert")
145     elif model_name == "BERT + Classifier":
146         emojis = predict_emoji_bert(tweet, model_name="bert_with_classifier")
147     elif model_name == "BERT + BiLSTM":
148         emojis = predict_emoji_bert(tweet, model_name="bidirectional_lstm")
149     elif model_name == "gloVe + BiLSTM":
150         emojis = predict_emoji(tweet, model_path="bidirectional_lstm_glove_model.keras")
151     elif model_name == "gloVe + LSTM + Attention":
152         emojis = predict_emoji(tweet, model_path="lstm_attention_glove_model.keras")
153     elif model_name == "gloVe + GRU":
154         emojis = predict_emoji(tweet, model_path="gru_glove_model.keras")
155     elif model_name == "gloVe + CNN":
156         emojis = predict_emoji(tweet, model_path="cnn_glove_model.keras")
157
158     top_emoji = emojis.iloc[int(emojis["prob"].idxmax())]["emoji"]
159
160     plot_emoji_bar_chart(emojis)
161     st.write("Tweet:", f"{tweet}{top_emoji}")
162     st.write("Predicted emoji:", top_emoji)

```

🔗 Overwriting app.py

```
1 !curl https://loca.lt/mytunnelpassword
```

🔗 35.243.211.254

```
1 !streamlit run app.py &>/content/logs.txt &
2 !npx localtunnel --port 8501
```

🔗 🌐: your url is: <https://dry-seals-tickle.loca.lt>

```

1
2
3
4
5
6
7
8
9
10

```