Building an NLP-Based Sentiment Analysis and Text Generation System Using Deep Learning

Course Title: Natural Language Processing

Student Name: Teberikov Nurasyl

Date: 04.03.2025

KBTU

1 Introduction

In Natural Language Processing (NLP), understanding and processing text effectively requires efficient representations and models. Word embeddings, Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs) play a crucial role in enabling deep learning models to handle sequential text data.

Traditional NLP approaches, such as one-hot encoding and TF-IDF, fail to capture semantic relationships between words and often result in high-dimensional, sparse representations. Word embeddings solve this problem by mapping words into dense, continuous vector spaces, where similar words are positioned closer together. Methods like Word2Vec, GloVe, and FastText generate embeddings that preserve semantic meaning, allowing models to understand word relationships and contextual similarities more effectively.[1]

Beyond word representation, processing sequential text data requires models that can retain context and learn from past information. Recurrent Neural Networks (RNNs) were introduced to handle such sequences by maintaining a hidden state, allowing the model to process words in context. However, RNNs suffer from the vanishing gradient problem, making it difficult to learn long-term dependencies.[2]

To overcome this limitation, Long Short-Term Memory networks (LSTMs) were developed.[1] LSTMs introduce gating mechanisms that help retain important information over long sequences, enabling models to understand context over extended text. An alternative to LSTMs, Gated Recurrent Units (GRUs), offer a simpler yet effective approach by reducing the number of gates, making them computationally efficient while still capturing long-range dependencies[2].

The purpose of this study is to build and compare different models based on word embeddings, RNNs, LSTMs, and GRUs, and evaluate their performance in sentiment classification and text generation tasks.

The project consists of the following key components:

- Text preprocessing using NLTK and spaCy to clean and prepare data.
- Training various word embeddings, including Word2Vec, GloVe, and FastText, to capture semantic relationships between words.
- Implementing and evaluating RNNs, LSTMs, GRUs, and Bidirectional LSTMs for sentiment classification.
- Generating text using an LSTM-based model to analyze its effectiveness in text generation tasks.

2 Data Preprocessing and Word Embeddings

2.1 Text Preprocessing with NLTK and spaCy

Effective text preprocessing is a crucial step in Natural Language Processing (NLP) to ensure that models can interpret and process textual data efficiently. In this project, we used NLTK and spaCy for tokenization, lemmatization, and filtering, which are essential for cleaning and structuring text data before feeding it into models. Tokenization involves splitting text into smaller units, such as words or subwords, making it easier for models to process.[3] Lemmatization converts words to their base or root form to standardize variations of the same word.[4]
Below is the relevant portion of the code demonstrating the text preprocessing pipeline, including tokenization, stopword removal, and lemmatization:

```
import nltk
import spacy
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Download resources
nltk.download('punkt')
nltk.download('stopwords')
nlp = spacy.load("en_core_web_sm")


# Sample text
data = pd.read_csv('/kaggle/input/covid19-tweets/covid19_tweets.csv')
data_text = data['text'].dropna().tolist()
data_text = data_text[:1000]

stop_words = set(stopwords.words('english'))

clean_text = []
for sentence in data_text:
    sentence = re.sub(r'[^a-zA-Z ]', '', sentence.lower())
    tokens = word_tokenize(sentence)
    lemmatized_tokens = [nlp(token)[0].lemma_ for token in tokens if token
        not in stop_words]
    clean_text.append(lemmatized_tokens)

clean_text[0:2]
```

2.2 Word Embeddings with Word2Vec, GloVe, and FastText

Word embeddings are numerical representations of words in a continuous vector space, capturing their meaning based on context and relationships with other words. Unlike traditional one-hot encoding or bag-of-words models, embeddings preserve semantic similarity, meaning that words with similar meanings have vectors that are close to each other in the embedding space.[5] In this project we used Word2Vec, FastText and GloVe:

```
# Word2Vec CBOW and Skip-gram
word2vec_cbow = Word2Vec(sentences=clean_text, vector_size=100, window=5,
    min_count=1, sg=0)
word2vec_sg = Word2Vec(sentences=clean_text, vector_size=100, window=5,
    min_count=1, sg=1)
```

```
# FastText
fast_text = FastText(sentences=clean_text, vector_size=100, window=5,
    min_count=1)
```

```
glove_embeddings = {}
path = '/kaggle/input/glove6b100dtxt/glove.6B.100d.txt'
with open(path, 'r', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], dtype='float32')
        glove_embeddings[word] = vector
```

Each of these embedding methods provides meaningful word representations, enhancing the ability of NLP models to understand and process natural language more effectively.

To better understand the structure of the word embeddings, Principal Component Analysis (PCA) was applied to reduce the high-dimensional word vectors into a two-dimensional space for visualization. This helps in observing how words with similar meanings cluster together and how different embedding techniques capture semantic relationships. The PCA visualizations of Word2Vec (CBOW) and FastTex embeddings (Figure 3 and Figure2) reveal how different words are positioned in the vector space based on their semantic relationships.

In both plots, the words ”people” and ”govt” appear close to each other. This suggests that these words frequently occur in similar contexts within the training corpus, indicating a strong relationship between governance and people.

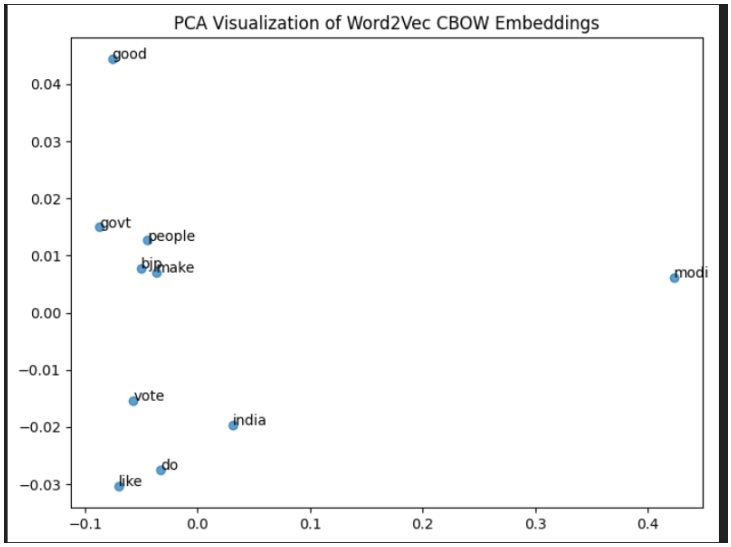

Figure 1: PCA Visualization of Word2Vec CBOW Embeddings

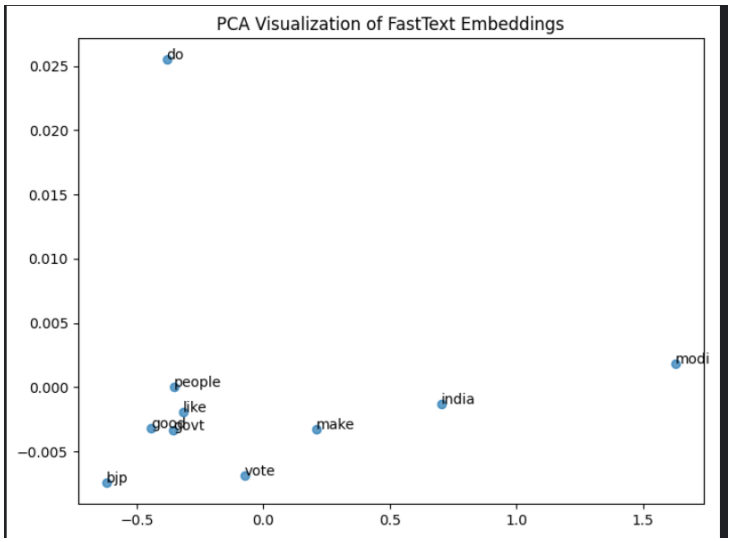

Figure 2: PCA Visualization of FastText Embeddings

2.3 Recurrent Neural Networks for Sentiment Analysis

A recurrent neural network or RNN is a deep neural network trained on sequential or time series data to create a machine learning (ML) model that can make sequential predictions or conclusions based on sequential inputs.[6] The recurrence is defined as:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b) \tag{1}$$

Sentiment analysis involves classifying text data (e.g., reviews, tweets) into categories based on sentiment polarity. A simple RNN can be used for this task as preprocessing, sequential input processing and sentiment prediction.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=10000, output_dim=32, input_length=100),
    tf.keras.layers.SimpleRNN(32, return_sequences=False),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
rnn_model = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=20, batch_size=32)
```

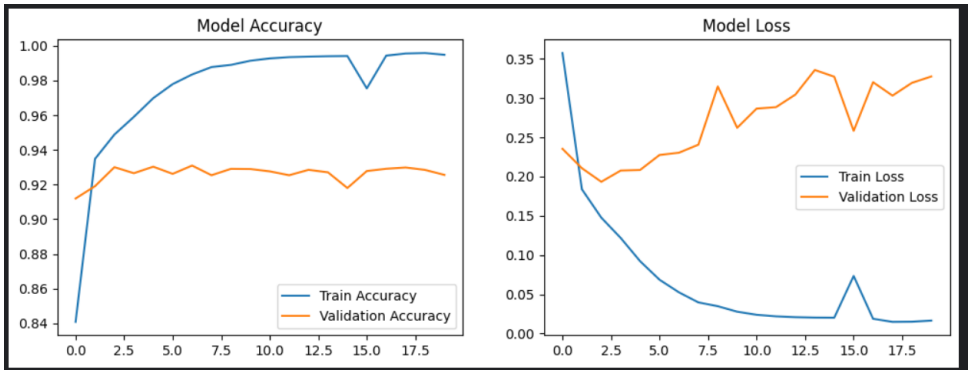

Figure 3: RNN training accuracy vs. loss curve

The Sequential model consists of an Embedding layer, a SimpleRNN layer, and a Dense layer for binary sentiment classification. While the SimpleRNN layer processes sequential data efficiently, training deep RNNs often presents challenges. One significant issue is the vanishing gradient problem, which occurs when gradients become extremely small during backpropagation through time.

To mitigate the vanishing gradient problem, gradient tracking was implemented during training using GradientTape in TensorFlow.

```
def get_gradients(epoch, logs):
    with tf.GradientTape() as tape:
        y_pred = model(X_train[:100])
        loss = tf.keras.losses.binary_crossentropy(tf.reshape(y_train[:100], (-1, 1)), y_pred)
    grads = tape.gradient(loss, model.trainable_variables)
    grads_norm = [tf.norm(g).numpy() for g in grads if g is not None]
    gradients.append(np.mean(grads_norm))

gradients = []
gradient_callback = tf.keras.callbacks.LambdaCallback(on_epoch_end=get_gradients)

```

```
model.fit(X_train, y_train, epochs=20, batch_size=32, callbacks=[
    gradient_callback])
```

As we can see in the Figure-4, the gradient values are relatively small across most epochs, except for a few sharp spikes. This suggests that for the majority of training, gradients remain close to zero, making it difficult for the network to update weights effectively.

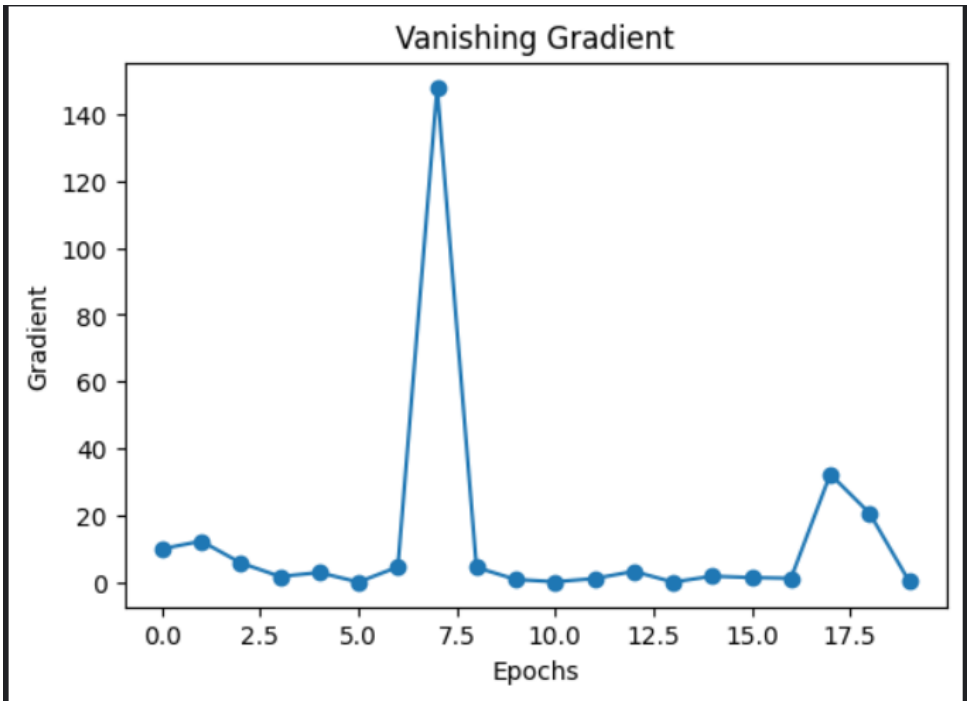

Figure 4: Gradient values over epochs

3 LSTM vs. GRU for Text Classification

LSTM and GRU are both types of recurrent neural networks designed to mitigate the vanishing gradient problem in standard RNNs. They achieve this through gating mechanisms that regulate the flow of information through the network.[7] However, they have some difference like: LSTM uses three gates (forget, input, and output) to control the amount of information retained or discarded, meanwhile GRU uses only two gates (reset and update) and combines hidden and cell states into a single state, making it computationally more efficient.

Text classification involves understanding sequential dependencies in textual data. Both LSTMs and GRUs are well-suited for this task because they can capture long-range dependencies.

```
#LSTM
model_lstm = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=10000, output_dim=32, input_length
        =100),
    tf.keras.layers.LSTM(32, return_sequences=False),
    tf.keras.layers.Dense(1, activation='sigmoid')
```

```
])

model_lstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

start_time = time.time()
lstm_res = model_lstm.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=20, batch_size=32)
print("implementation of LSTM model", time.time() - start_time)

#GRU
model_gru = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=10000, output_dim=32, input_length=100),
    tf.keras.layers.GRU(32, return_sequences=False),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model_gru.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

start_time = time.time()
gru_res = model_gru.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=20, batch_size=32)
print("implementation of GRU model", time.time() - start_time)
```

The comparison of LSTM and GRU performance in text classification is shown in the accuracy plot (Figure -5). From the accuracy plot, both LSTM and GRU achieve high training accuracy, nearing 100% as epochs increase. However, the validation accuracy for both models plateaus around 96%, with slight fluctuations, indicating a possible overfitting trend.

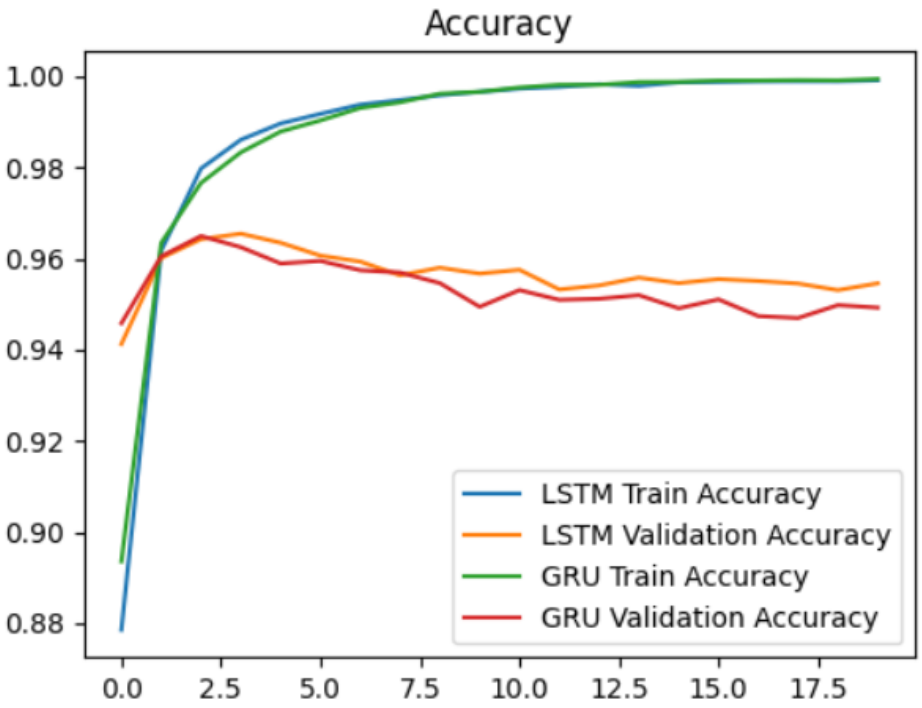

Figure 5: LSTM vs. GRU accuracy comparison

The loss plot (Figure - 6) further confirms this observation. The training loss for both models decreases steadily, suggesting effective learning on the training data. However, the validation loss for both LSTM and GRU starts increasing after a few epochs, indicating that the models struggle to generalize well to unseen data. Notably, the GRU validation loss shows a higher degree of fluctuation compared to LSTM, which suggests that GRU may be slightly more sensitive to variations in the validation dataset.

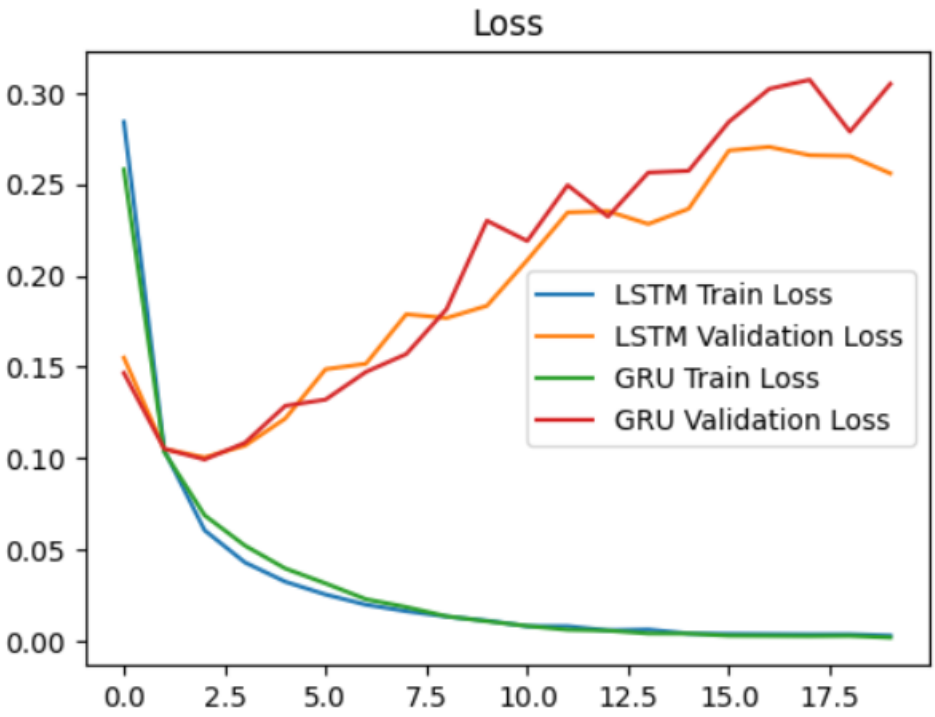

Figure 6: LSTM vs. GRU loss comparison

4 Text Generation with LSTM

We randomly selected 50 elements from our dataset to create a subset for text generation. After sampling, we tokenized the text using the Tokenizer class, limiting the vocabulary size to 10,000 words. Next, we generated sequential training data by progressively expanding sequences within each text sample. These sequences were then converted into numerical form using the tokenizer, ensuring that all text data was represented in a consistent manner. To handle variable sequence lengths, we applied padding using pad_sequences, aligning all sequences to the maximum length found in the dataset. The padded sequences were then split into input (X_textgen) and target (y_textgen), with the target variable one-hot encoded for categorical classification.

To train the text generation model, we built a sequential neural network using LSTM layers. The model starts with an Embedding layer, which maps input words into dense vector representations of size 50. The second LSTM layer extracted high-level dependencies and passed its output to a Dense layer with 100 neurons and ReLU activation. Finally, a softmax output layer predicted the next word by computing probabilities over the entire vocabulary. We compiled the model using the categorical cross-entropy loss function and the Adam optimizer, with accuracy as the evaluation metric. The training process ran for 100 epochs with a batch size of 64, ensuring sufficient learning of word relationships from the dataset.

Once the model was trained, we used it for text generation by providing a seed phrase. The

model predicted the next word based on the input, appended it to the sequence, and repeated the process iteratively to generate coherent text.

```
sampled_texts = data['clean_text'].sample(n=50, random_state=42).values
sampled_texts[0:2]

tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(sampled_texts)
total_words = len(tokenizer.word_index) + 1

sequence_data = []
for text in sampled_texts:
    words = text.split()
    for i in range(1, len(words)):
        sequence_data.append(words[:i+1])

sequences = tokenizer.texts_to_sequences([" ".join(seq) for seq in
    sequence_data])

sequences = [seq for seq in sequences if len(seq) > 1]


max_sequence_length = max(len(seq) for seq in sequences)
sequences = pad_sequences(sequences, maxlen=max_sequence_length, padding='
    pre')

X_textgen, y_textgen = sequences[:, :-1], sequences[:, -1]

y_textgen = tf.keras.utils.to_categorical(y_textgen, num_classes=
    total_words)

textgen_model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=len(tokenizer.word_index) + 1,
        output_dim=50, input_length=max_sequence_length - 1),
    tf.keras.layers.LSTM(100, return_sequences=True),
    tf.keras.layers.LSTM(100),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(len(tokenizer.word_index) + 1, activation='
        softmax')
])

textgen_model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
textgen_model.fit(X_textgen, y_textgen, epochs=100, batch_size=64)

seed_texts = ["combination", "india", "trump"]

```

```
n_words = 20

for seed_text in seed_texts:
    print('input words', seed_text)

    for _ in range(n_words):
        tokenized_input = tokenizer.texts_to_sequences([seed_text])[0]
        tokenized_input = pad_sequences([tokenized_input], maxlen=
            max_sequence_length - 1, padding='pre')
        predicted_index = np.argmax(textgen_model.predict(tokenized_input),
             axis=-1)[0]

        output_word = ""
        for word, index in tokenizer.word_index.items():
            if index == predicted_index:
                output_word = word
                break

        seed_text += " " + output_word

    print('result text', seed_text)
    print("-")
```

To evaluate the performance of our trained LSTM model, we used it to generate text based on given seed words. Figure-7 presents the results of text generation for three different seed words: ”combination”, ”india”, and ”trump”. The model iteratively predicted the next word in the sequence, progressively forming longer and more contextually coherent sentences.

```
result text combination 2019 election nda govt takes overpeople will get hold all opposition party netas who abused armed forces modibjp and india
result text india cheer freedom reply need ypur help very urgent please regards rohan saxena ph9927290097 from bazpur uttarakhand they that truth coots
result text trump 2019 election nda govt takes overpeople will get hold all opposition party netas who abused armed forces modibjp and india
```

Figure 7: Text generated by the LSTM

5 Bidirectional LSTM for Improved Performance

To enhance the text classification model, we replaced the standard LSTM layer with a Bidirectional LSTM (BiLSTM) layer. Unlike a conventional LSTM, which processes input sequences in a single direction (left to right), a BiLSTM processes sequences in both forward and backward directions[8]. This modification allows the model to capture dependencies from both past and future contexts, leading to improved understanding of text sequences.

```
model_bilstm = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=10000, output_dim=32, input_length
        =100),
```

```
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences
        =False)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model_bilstm.compile(optimizer='adam', loss='binary_crossentropy', metrics
    =['accuracy'])

start_time = time.time()
bilstm_res = model_bilstm.fit(X_train, y_train, validation_data=(X_test,
    y_test), epochs=20, batch_size=32)
print('implementation of BI LSTM model',time.time() - start_time)
```

To evaluate the effectiveness of the BiLSTM model compared to the standard LSTM, we analyzed their training accuracy, validation accuracy, training loss, and validation loss across multiple epochs. Figure-8 presents a comparison of accuracy (left) and loss (right) for both models.

From the accuracy plot, we observe that both models achieved similar validation accuracy, with BiLSTM slightly outperforming LSTM in the early epochs. However, the BiLSTM model exhibits higher training accuracy, indicating that it captures more patterns from the data but may also be prone to overfitting.

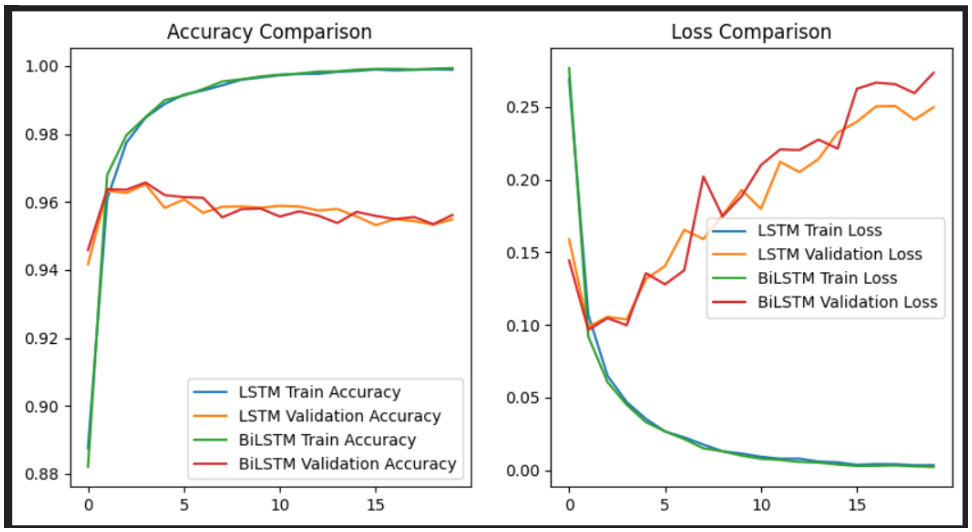

Figure 8: Comparison of Bidirectional LSTM and LSTM model performance

6 Conclusion

This study explored various deep learning techniques for Natural Language Processing, focusing on sentiment analysis and text generation. We implemented and compared multiple architectures, including Recurrent Neural Networks, Long Short-Term Memory networks, Gated Recurrent Units (GRU), and Bidirectional LSTM, to assess their effectiveness in processing se-

quential text data. LSTMs and GRUs outperformed traditional RNNs by addressing long-range dependency issues, while BiLSTM improved contextual understanding. Sentiment classification models achieved high accuracy but showed signs of overfitting. The LSTM-based text generator produced coherent sequences, demonstrating its capability in sequence modeling. Despite the results, problems such as over-fitting and computational efficiency remain. Future work could explore transformer-based models for improved performance.

References

[1] Ajay Halthor. Word2vec, glove, and fasttext, explained. 6 2023.

[2] Egor Howell. Gated recurrent units (gru) – improving rnns. 6 2024.

[3] Sabrina J. Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp. 12 2021.

[4] Niklas Lang. Stemming vs. lemmatization in nlp. 2 2022.

[5] Harsh Vardhan. A comprehensive guide to word embeddings in nlp. 7 2024.

[6] Cole Stryker. What is a recurrent neural network? 10 2024.

[7] Prudhviraju Srivatsavaya. Lstm vs gru. 7 2023.

[8] Anishnama. Understanding bidirectional lstm for sequential data processing. 5 2023.