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

Course Title: Natural Language Processing

Student Name: Kumarkhanova Ayazhan

Date: 04.03.2025

KBTU

1 Introduction

Sentiment analysis is a process of extracting and understanding the sentiments defined in the text document. The explosion of data in the various social media channels like twitter, facebook, and linkdin has given consumer new way of expressing their opinion on a particular product, person and places. [1] Text generation is a key component of language translation, chatbots, question answering, summarization, and several other applications that people interact with everyday. Building language models using traditional approaches is a complicated task that needs to take into account multiple aspects of language, including linguistic structure, grammar, word usage, and reasoning, and thus requires non-trivial data labeling efforts.[2]

Deep learning has evolved many algorithms in the field of NLP like Recurrent Neural Networks (RNNs) (for sequence modeling). Since RNNs contain internal memory due to which it is able to remember the previous input as well as current input that makes sequence modeling tasks lot easier. [3, 4] LSTMs inherits the same architecture as RNNs, without hidden state. The memory units in LSTMs are called cells that take the combination of previous state and current input as input. These cells actually decide what to keep in memory and what to eliminate. [5, 3]

GRU is another extension of standard RNNs (Cho et al., 2014) which modifies LSTM architecture with a gating network, which generates signals that control the present input and previous memory to update the current activation and current network state. GRUs layers widely used for the generation of text. [3, 6] **The purpose** of this project is to build and compare different models for sentiment classification and text generation. By evaluating the performance of various models, we aim to compare different architectures for these tasks. The models are assessed based on accuracy, loss, and their ability to capture meaningful representations from text.

2 Data Preprocessing and Word Embeddings

Text Preprocessing We applied the following preprocessing steps using spaCy and nltk libraries. We used the spaCy en_core_web_sm model to tokenize text into words, it splits text into meaningful units. Then convert them to lemmas or to their base. For example: ”turned” -¿ ”turn”. And removed stopwords (using NLTK’s stopword list) and punctuation.

```
import spacy
import nltk
from nltk.corpus import stopwords

nlp = spacy.load("en_core_web_sm")
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))
```

```
8
9  def preprocess_text(text):
10     doc = nlp(text.lower())
11     tokens = [token.lemma_ for token in doc if token.is_alpha and
           token.text not in stop_words]
12     return tokens
13
14 df["processed_review"] = df["Review"].apply(preprocess_text)
15
16 df["processed_review"].iloc[0]
```

Word Embeddings At the beginning of any NLP tasks, when we try to learn joint probability functions of language models there is the problem of dimensionality, so it is always necessary to understand the distributed representation of text in a low dimensional space. Word embedding [3] is the mapping of a discrete categorical variable to a vector of continuous numbers. In the context of Artificial Neural Networks (ANNs), embeddings are low dimensional learned continuous vector representation of discrete variables. Word embeddings actually follow the distributional hypothesis where words having similar meanings occur in a similar context. Thus these vectors try to capture the characteristics of words that are closer to each other. Distributional vectors actually capture similarity (cosine similarity) between words. The significance of word embeddings in the field of deep learning turns into noticeable by considering the number of researchers in the field. One such research in the field of word embeddings led by Google prompted the advancement in the group of related techniques or algorithms commonly known to as Word2Vec. [2]

```
1  from gensim.models import Word2Vec
2
3  sentences = df["processed_review"].tolist()
4
5  # CBOW model
6  cbow_model = Word2Vec(sentences, vector_size=100, window=5,
       min_count=5, sg=0)
7  # Skip-gram model
8  skipgram_model = Word2Vec(sentences, vector_size=100, window=5,
       min_count=5, sg=1)
9
10 # get vector for a word
11 w2v_vector_cbow = cbow_model.wv["checkin"]
12 print(w2v_vector_cbow)
13
14 w2v_vector_skipgram = skipgram_model.wv["checkin"]
15 print(w2v_vector_skipgram)
```

The problem of Word2Vec is that it relies on local context of sentences, which means it captures only semantic information of language. However, this can be suboptimal sometimes. Glove (Global Vectors) on the other hand, captures both the global context as well as the local context of vocabulary while changing words to vectors. However, each type of measurement has its own advantages. For example, in the case of analogy tasks, Word2Vec does very well. The Glove works on the co-occurrence of words.

```
glove_path = kagglehub.dataset_download("danielwillgeorge/
   glove6b100dtxt")
glove_path = '/kaggle/input/glove6b100dtxt/glove.6B.100d.txt'

glove_embeddings = {}

with open(glove_path, "r", encoding="utf-8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.array(values[1:], dtype="float32")
        glove_embeddings[word] = vector

# get vector
glove_vector = glove_embeddings["checkin"]
print(glove_vector)
```

Another method of representing words into vectors is FastText which is an extension of Word2Vec. This method represents each word as an n-gram of characters rather than representing words directly.[2, 3, 7]

```
sentences = df["processed_review"].tolist()
fasttext_model = FastText(sentences, vector_size=100, epochs=10)

# get vector
fasttext_vector = fasttext_model.wv["checkin"]
print(fasttext_vector)
```

To compare and evaluate these embeddings we used cosine similarity. It measures the closeness of word vectors. Higher similarity indicates better word associations.

```
from sklearn.metrics.pairwise import cosine_similarity

# cosine similarity
similarity_w2v_glove = cosine_similarity([w2v_vector_cbow], [
   glove_vector])[0][0]
```

```
similarity_w2v_fasttext = cosine_similarity([w2v_vector_cbow], [
    fasttext_vector])[0][0]
similarity_glove_fasttext = cosine_similarity([glove_vector], [
    fasttext_vector])[0][0]

print(f"Word2Vec and GloVe Similarity: {similarity_w2v_glove:.4f}")
print(f"Word2Vec and FastText Similarity: {similarity_w2v_fasttext
    :.4f}")
print(f"GloVe and FastText Similarity: {similarity_glove_fasttext:.4
    f}")
```

Similarity results: Word2Vec and GloVe Similarity: -0.0283 Word2Vec and FastText Similarity: 0.2296 GloVe and FastText Similarity: -0.0933

So now words are represented as 100-D vectors. To visualize these embeddings, we need to reduce their dimensionality but preserve their structure. Two popular techniques for this are Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) [8].

Figures 1 and 2 show the embedding visualizations.

t-SNE is a widely used non-linear dimensionality reduction technique for visualizing high-dimensional data with clear and perfect separation, in the two (or three) dimensional plane. First, a probability distribution is estimated among the pairs of high-dimensional data points in such a way that similar objects are assigned a high probability of being selected and dissimilar points are assigned small probability of being chosen. Second, t-SNE assigns a uniform probability distribution model in the low-dimensional map. [9]

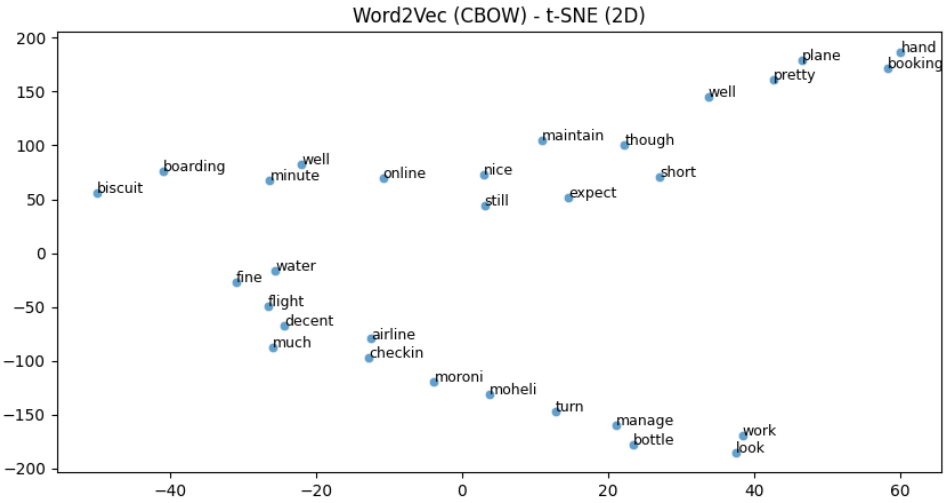

Figure 1: t-SNE visualization of Word2Vec embeddings

PCA extract the important information from the table, to represent it as a set of new orthogonal variables called principal components, and to display the pattern of similarity of the

observations and of the variables as points in maps. [10]

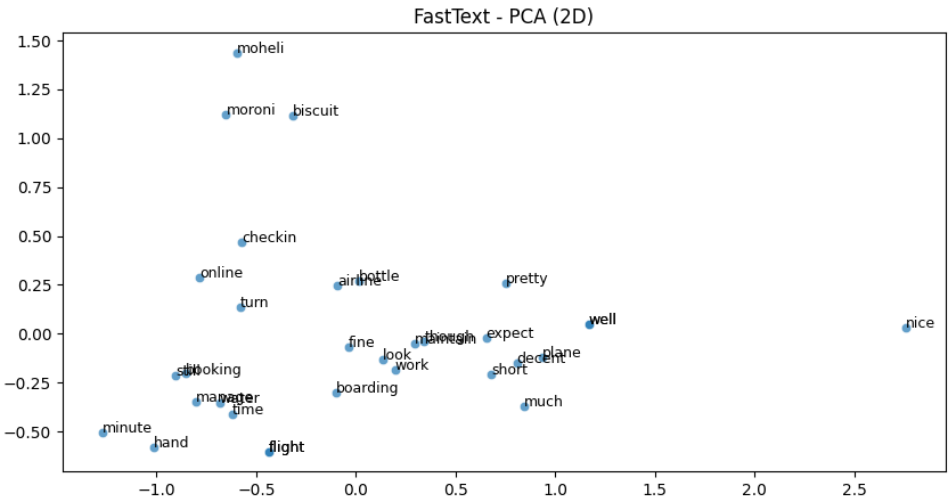

Figure 2: PCA visualization of FastText embeddings

3 Recurrent Neural Networks for Sentiment Analysis

In ANNs all inputs are independent of each other, but in RNNs there is dependency among inputs. The output at particular time-step "t" is given by the equation:

$$h_t = \sum X_t W_t + U h_{t-1} + b \tag{1}$$

h_{t-1} is previous output and X_t is current input and W_t represent weight at time step t, also U represent weight associated with output h_{t-1} and b represents bias term. At each iteration t the input X_t is fed to the network where the h_t is calculated based on previous output h_{t-1}. [3] The output at any time step does not only depend on current input but also on the output generated at previous time steps, which makes it good of tasks like language generation, language translation, sentiment analysis, etc.

```
model = Sequential([
    Embedding(input_dim=max_words, output_dim=100, input_length=
        max_len),
    SimpleRNN(128, activation='tanh', return_sequences=True, dropout
        =0.2),
    SimpleRNN(64, activation='tanh', return_sequences=False, dropout
        =0.2),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

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

def compute_gradients(model, x_sample, y_sample):
    with tf.GradientTape() as tape:
        y_pred = model(x_sample, training=True)
        loss = tf.keras.losses.binary_crossentropy(tf.reshape(
            y_sample, (-1, 1)), y_pred)
        loss = tf.reduce_mean(loss)
    grads = tape.gradient(loss, model.trainable_variables)
    return [tf.norm(g).numpy() if g is not None else 0 for g in
        grads]

gradient_history = []
history = []
for epoch in range(20):
    hist = model.fit(X_train, y_train, epochs=1, batch_size=32,
        validation_data=(X_test, y_test), verbose=1)

    gradients = compute_gradients(model, X_train[:32], y_train[:32])
    gradient_history.append(gradients)
    history.append(hist.history)

plt.figure(figsize=(10, 5))
for i in rnn_layer_indices:
    plt.plot(range(1, 21), gradient_history[:, i], label=f"RNN Layer
        {i+1}")

plt.xlabel("Epoch")
plt.ylabel("Gradient Norm")
plt.title("Gradient Norm Evolution Across RNN Layers")
plt.legend()
plt.grid(True)
plt.show()
```

Training is performed for 20 epochs with batch size 32. 3 shows the accuracy and loss curves by epoch. The training accuracy increases, but the validation accuracy remains unchanged and eventually drops, indicating that the model is overfitting. The training loss decreases significantly, while the validation loss oscillates and increases over time, confirming that the model does not generalize well to unseen data.

4 shows the gradient normals for the different RNN layers over the epochs. The third RNN

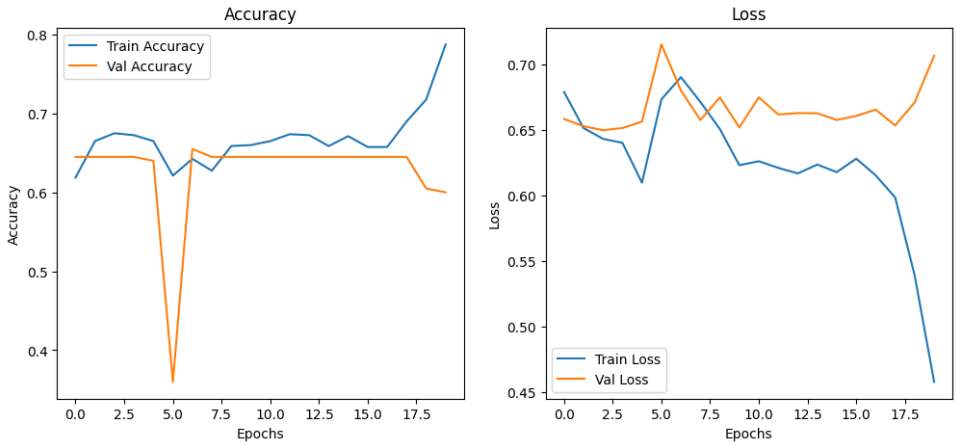

Figure 3: RNN training accuracy vs. loss curve

layer shows an extreme gradient spike around epoch 7, followed by exploding gradients. In the subsequent epochs, the gradients are close to zero, confirming the vanishing gradient problem. Using architectures such as LSTM or GRU instead of RNN can help with this problem by keeping a better gradient flow.

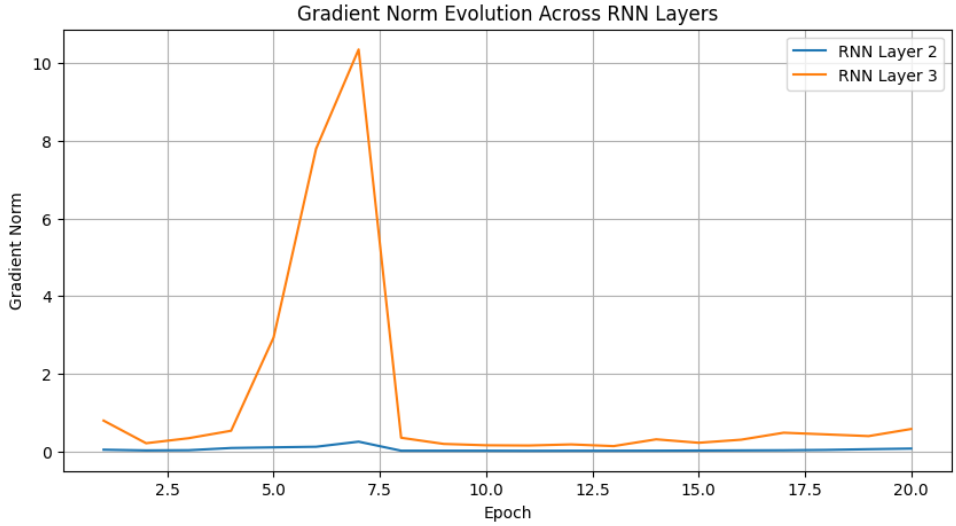

Figure 4: Gradient values over epochs (vanishing gradient problem)

4 LSTM vs. GRU for Text Classification

LSTM and GRU are both designed to address the vanishing gradient problem in standard RNN by incorporating gating mechanisms. LSTM has a more complex structure with three gates: input, forget, and output gates, allowing fine-grained control over memory updates.

```
lstm_model = Sequential([
    Embedding(input_dim=max_words, output_dim=100, input_length=
        max_len),
```

```
    LSTM(128, activation='tanh', return_sequences=True, dropout=0.2),
    LSTM(64, activation='tanh', return_sequences=False, dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

lstm_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
start_time = time.time()
history_lstm = lstm_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test), verbose=1)
lstm_time = time.time() - start_time
```

GRU, on the other hand, has a simpler architecture with only two gates: reset and update, making them computationally more efficient.

```
gru_model = Sequential([
    Embedding(input_dim=max_words, output_dim=100, input_length=max_len),
    GRU(128, activation='tanh', return_sequences=True, dropout=0.2),
    GRU(64, activation='tanh', return_sequences=False, dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

gru_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
start_time = time.time()
history_gru = gru_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test), verbose=1)
gru_time = time.time() - start_time
```

According to 5 LSTM might be better for this task, as it shows higher validation accuracy.

Training loss (see 6) for both models decreases steadily, suggesting that both are learning effectively. Validation loss for LSTM and GRU increase over time (may be overfitting).

Figure 5: LSTM vs. GRU accuracy comparison

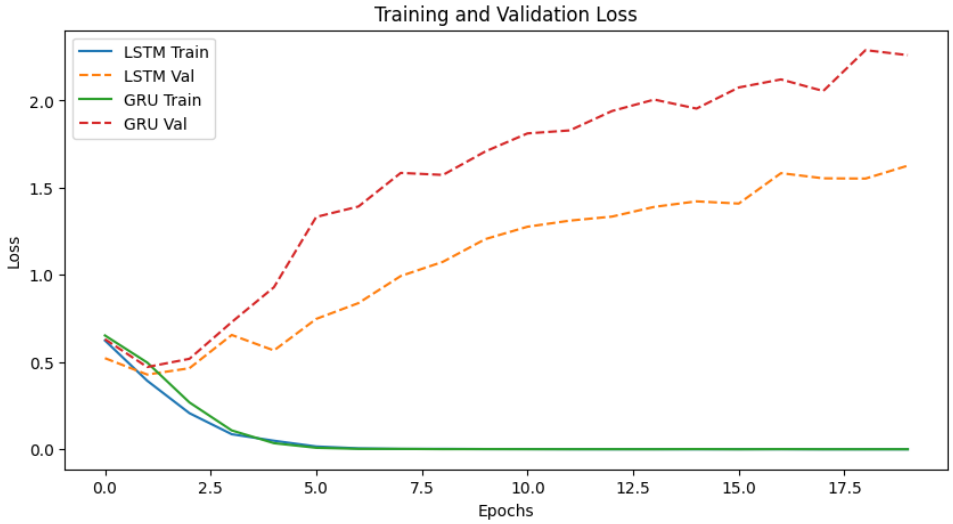

Figure 6: LSTM vs. GRU loss comparison

5 Text Generation with LSTM

The LSTM-based text generation model consists of an embedding layer, two stacked LSTM layers (256 and 128 units, respectively), followed by two dense layers. The final dense layer uses a softmax activation to predict the next word. The model is trained using categorical cross-entropy loss and optimized with Adam.

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(reviews)
total_words = len(tokenizer.word_index) + 1

input_sequences = []
for review in reviews:
```

```
    token_list = tokenizer.texts_to_sequences([review])[0]
    for i in range(1, len(token_list)):
        input_sequences.append(token_list[:i+1])

# padding
max_seq_length = max(len(seq) for seq in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=
   max_seq_length, padding='pre')

# X (input) and y (next word)
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)

model = Sequential([
    Embedding(total_words, 100, input_length=max_seq_length-1),
    LSTM(256, return_sequences=True),
    LSTM(128),
    Dense(128, activation='relu'),
    Dense(total_words, activation='softmax')
])

model.compile(loss='categorical_crossentropy', optimizer='adam',
   metrics=['accuracy'])
model.fit(X, y, epochs=70, verbose=2)
```

LSTM model generates text by predicting the next word based on the given seed text. For example:

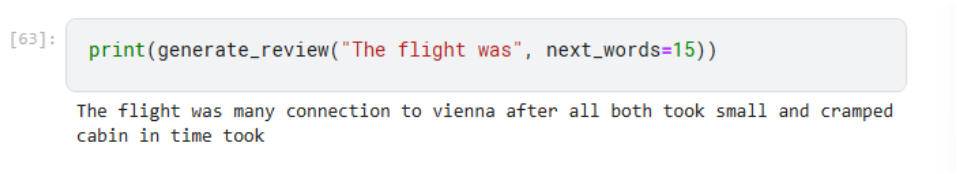

Figure 7: Text generated by the LSTM model based on the Airline reviews seed text

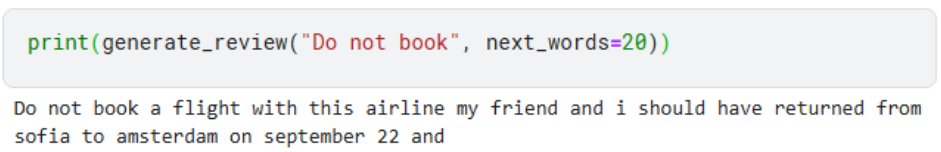

Figure 8: Generated text from a different seed text

6 Bidirectional LSTM for Improved Performance

Bidirectional LSTMs are an extension of the described LSTM models in which two LSTMs are applied to the input data. In the first round, an LSTM is applied on the input sequence (forward layer). In the second round, the reverse form of the input sequence is fed into the LSTM model (backward layer). Applying the LSTM twice leads to improve learning long-term dependencies and thus consequently will improve the accuracy of the model. [11, 12]

```
# Bi-LSTM
bilstm_model = Sequential([
    Embedding(input_dim=max_words, output_dim=100, input_length=
       max_len),
    Bidirectional(LSTM(128, activation='tanh', return_sequences=True
       , dropout=0.2)),
    Bidirectional(LSTM(64, activation='tanh', return_sequences=False
       , dropout=0.2)),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

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

start_time = time.time()
history_bilstm = bilstm_model.fit(X_train, y_train, epochs=20,
   batch_size=32, validation_data=(X_test, y_test), verbose=1)
bilstm_time = time.time() - start_time
```

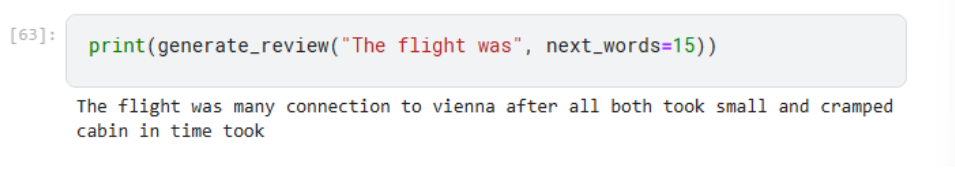

Figure 9: Text generated by the LSTM model based on the Airline reviews seed text

In theory, BiLSTM has potentially better performance in NLP tasks, but in my task the improvement isn't that big as you can see from 10. Since performance is similar, a regular LSTM is more efficient because BiLSTM doubles computation.

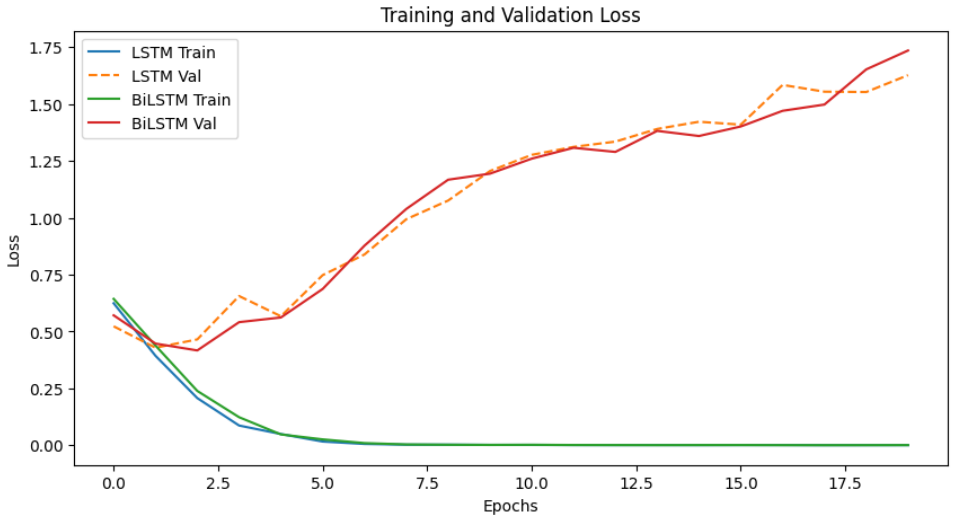

Figure 10: Comparison of Bidirectional LSTM and LSTM model performance

7 Conclusion

We trained and evaluated multiple architectures, like LSTM, GRU, and BiLSTM, for text generation and classification tasks. All models achieved similar training accuracy, but GRU, being simpler, trained faster. However, LSTM performed a bit better than others. BiLSTM model showed only minimal improvement over the standard LSTM, suggesting that bidirectional context did not significantly enhance performance for this dataset. The main problem of all models was a sign of overfitting. To achive better perfomance, future improvements could include hyperparameter tuning, better regularization to reduce overfitting, and attention mechanisms to enhance sequence understanding

8 References

References

[1] T. K. Shivaprasad and Jyothi Shetty. Sentiment analysis of product reviews: A review. In *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 298–301, 2017.

[2] Asli Celikyilmaz, Elizabeth Clark, and Jianfeng Gao. Evaluation of text generation: A survey, 2021.

[3] Touseef Iqbal and Shaima Qureshi. The survey: Text generation models in deep learning. *Journal of King Saud University - Computer and Information Sciences*, 34(6, Part A):2515–2528, 2022.

[4] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Sanidhya Mangal, Poorva Joshi, and Rahul Modak. Lstm vs. gru vs. bidirectional rnn for script generation. *arXiv preprint arXiv:1908.04332*, 2019.

[7] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405*, 2017.

[8] Jyoti Pareek and Joel Jacob. Data compression and visualization using pca and t-sne. In *Advances in Information Communication Technology and Computing: Proceedings of AICTC 2019*, pages 327–337. Springer, 2021.

[9] Mujtaba Husnain, Malik Muhammad Saad Missen, Shahzad Mumtaz, Muhammad Muzzamil Luqman, Mickaël Coustaty, and Jean-Marc Ogier. Visualization of high-dimensional data by pairwise fusion matrices using t-sne. *Symmetry*, 11(1), 2019.

[10] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[11] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[12] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. The performance of lstm and bilstm in forecasting time series. In *2019 IEEE International conference on big data (Big Data)*, pages 3285–3292. IEEE, 2019.