UNIVERSITY OF WOLVERHAMPTON

Artificial Intelligence and Machine Learning

(6CS012)

Sentiment Analysis of IMDb Movie Reviews Using RNN, LSTM, and Word2Vec Embeddings

Assessment Report II

Full name : Ananda Neupane

Student ID : 2329810

Group Name : L6CG2

Submission date : March 10, 2025

Tutor : Shiv Kumar yadav

Abstract

Inside this project, I work on a sentiment classification task involving IMDb movie reviews using a DL based approach. This project centers on understanding how models like “Recurrent Neural Networks (RNN)”, “Long Short-Term Memory (LSTM) networks”, and pretrained Word2Vec embeddings perform when applied to textual sentiment prediction. The ability to automatically detect sentiment from written language is valuable for various real-world applications, including content recommendation and market research.

To approach this problem, I build and train three models: a basic RNN, a more advanced LSTM, and an LSTM enhanced with semantic knowledge from Word2Vec embeddings. All models are trained using a clean and preprocessed IMDb movie review dataset. The data is tokenized, lemmatized, and transformed into vectorized form for training. My results show that the LSTM model outperforms the Simple RNN in terms of accuracy and overall performance, achieving 75% accuracy versus 66%. Adding Word2Vec embeddings maintains that accuracy while improving class balance in metrics like recall and F1-score.

I find that using LSTM-based architecture leads to better generalization, especially when semantic-rich embeddings are incorporated. I also observe subtle signs of training variability, particularly when working with fixed Word2Vec weights. For future improvements, I aim to integrate attention mechanisms and explore transformer-based models like BERT, as well as refine deployment tools such as the Streamlit GUI I develop for live sentiment prediction.

Table of Contents

Table of Figures

1. Introduction

Sentiment classification of movie reviews is a meaningful task that can help users make informed viewing decisions and provide feedback to filmmakers. In this work, I aim to develop models that can effectively distinguish between positive and negative sentiments. Since movie reviews are sequential in nature, I choose to employ RNN and LSTM models, which are best fit for capturing dependencies in sequence data. I also investigate how pretrained Word2Vec embeddings can enhance these models by providing richer word representations. Through this report, I explore how these deep learning methods perform on the sentiment analysis task and share insights from my findings.

2. Dataset

1. Source:

Primary source of this dataset can be found in Kaggle which contains IMDB Movie Reviews dataset provided in CSV format. I get this custom extracted dataset directly from my tutor which was primarily divided into three files of training, validation, and test subsets.

2. Size:

The dataset includes 35,000 rows data for training, 5,000 row data for validation, and 10,000 data for testing. Each review contains target label as either positive (1) or negative (0), forming a binary classification task. The file size of the train dataset is nearly 9.4 MB meanwhile the validation dataset of 1.4 MB and lastly test is of 2.8 MB approximately.

3. Feature:

Each record consists of a textual movie review and a sentiment label. An additional column is generated for storing cleaned text used during preprocessing.

4. Preprocessing:

I carry out several text-cleaning operations to enhance the quality and relevancy of the input data:

I. Convert all characters in the text to lowercase (e.g., "Amazing Film!" → "amazing film")
II. Expand contractions for standardization (e.g., "didn't" → "did not")
III. Remove URLs, mentions (e.g., @user), and hashtags (e.g., "Check out https://... #Awesome" → "check out")
IV. Strip out special characters and numeric digits (e.g., "$5.99" → "")
V. Eliminate common English stop words utilizing the NLTK's stop word list (e.g., remove words like "is", "the", "on")
VI. Apply lemmatization using WordNetLemmatizer to decompose words into their base forms (e.g., "running" → "run")
VII. Tokenize and join words back into cleaned sentences (e.g., ["this", "film", "good"] → "this film good")

This preprocessing pipeline ensures that only meaningful textual information is passed into the models, reducing noise and improving classification performance.

3. Methodology

3.1. Text Preprocessing:

I begin by cleaning the raw review text. I strip out noise such as punctuation, HTML tags, and irrelevant symbols. After tokenizing the cleaned text, I remove stopwords and lemmatize the remaining tokens to standardize them.

3.2. Model Architecture

1. Simple RNN:

The Simple RNN model is developed to handle sequential data by keeping a hidden state that holds previous word information while processing the current word. It initiates with an embedding layer that transforms each word into a dense vector. This is further

transfer into a Simple RNN layer containing 64 units that helps capture sequential dependencies. Finally, a dense sigmoid-activated output layer provides binary classification. While efficient for short sequences, Simple RNNs often struggle with long-term dependencies, which can limit their performance on lengthy reviews.

2. LSTM:

To address the shortcomings of the Simple RNN, I use a “Long Short-Term Memory” (LSTM) network. The LSTM model includes memory cells along with gating mechanisms that allow it to retain and manage information across longer sequences. This makes it more effective in capturing context and dependencies in movie reviews. The architecture also consists of an embedding layer followed by an LSTM layer with 64 units and a final sigmoid-activated output layer for binary classification. It provides better stability during training and improves overall accuracy.

3. LSTM with Word2Vec Embeddings

The most advanced model I use combines the LSTM architecture with pretrained Word2Vec embeddings. These embeddings, sourced from the Google News dataset, provide semantic word representations based on large-scale co-occurrence statistics. I align these embeddings with the vocabulary in my dataset and feed them into a fixed, non-trainable embedding layer. This enables the model to benefit from external linguistic knowledge, leading to faster convergence and enhanced accuracy. This model significantly improves performance by enriching the representation of input data.

3.3. Training Configuration

1. Loss Function:

For the binary classification task, I apply the binary cross-entropy loss function. It calculates the divergence among the predicted probability and the actual class label. This method is highly appropriate for two-class problems and helps refine the model's prediction accuracy by assigning heavier penalties to incorrect outputs.

2. Optimizer:

I train the models using the Adam optimizer, a widely adopted algorithm famous about its adaptive learning rate and momentum properties. It merges the strengths of both RMSProp and SGD with momentum, allowing for fast and stable convergence. Its efficiency and easy to use algorithms make it best fit for training models on natural language data.

3. Hyperparameters:

For training phase, I assign the learning rate to 0.0001, allowing gradual weight updates to prevent overshooting minima. The batch size is fixed at 64, enabling balanced memory usage and stable gradient updates. I run training for up to 50 epochs, though early stopping is employed to terminate training when no further validation improvements are observed until patience value of 6. Additionally, model checkpointing is used to retain the version of the model that achieves the highest validation accuracy.

4. Experiments and Results

4.1. RNN vs. LSTM Performance

To evaluate the difference in effectiveness between the RNN and LSTM architectures, I analyze their final accuracy and training time. The Simple RNN model resulted an accuracy of 66% over the test set, with a training time of 193.45 seconds. In contrast, the LSTM model reached an accuracy of 75%, improving significantly over RNN, with a slightly lower training time of 175.77 seconds. This demonstrates the LSTM's capability to learn longer dependency over sequential data and achieve better generalization.

4.2. Computational Efficiency

From a computational perspective, the RNN model took longer to converge and demonstrated instability in learning over long sequences. Despite the LSTM model having a more complex structure, it proved to be more computationally efficient in training due to quicker convergence and improved accuracy per epoch. All experiments were performed on Google Colab utilizing GPU acceleration, which was essential for reducing training time and memory usage.

4.3. Training with Different Embeddings

I also compare the impact of different embedding strategies. When using randomly initialized embeddings with the LSTM model, the model reached 75% accuracy. However, substituting random embeddings with pretrained Word2Vec embeddings marginally improved the model's ability to capture semantic relationships, resulting in another 75% accuracy but with different precision/recall trade-offs. Word2Vec-trained models had better generalization and slightly faster convergence during training (184.02 seconds).

4.4. Model Evaluation

For a comprehensive assessment, I evaluate the models with four different metrics of accuracy, precision, recall, and F1-score.

1. Simple RNN Classification Report:

I. Accuracy: 66%

II. Precision: 0.64 (positive), 0.68 (negative)

III. Recall: 0.73 (positive), 0.59 (negative)

IV. F1-Score: 0.68 (positive), 0.63 (negative)

2. LSTM Classification Report:

I. Accuracy: 75%

II. Precision: 0.77 (positive), 0.74 (negative)

III. Recall: 0.74 (positive), 0.77 (negative)

IV. F1-Score: 0.75 (positive), 0.76 (negative)

3. LSTM with Word2Vec Classification Report:

I. Accuracy: 75%

II. Precision: 0.74 (positive), 0.77 (negative)

III. Recall: 0.79 (positive), 0.71 (negative)

IV. F1-Score: 0.76 (positive), 0.74 (negative)

Each evaluation demonstrates the performance gains when transitioning from RNN to LSTM and the subtle but meaningful improvement gained with Word2Vec integration.

4.5. Accuracy and Loss Visualization

The following diagrams interprets the training and validation accuracy and loss for each model:

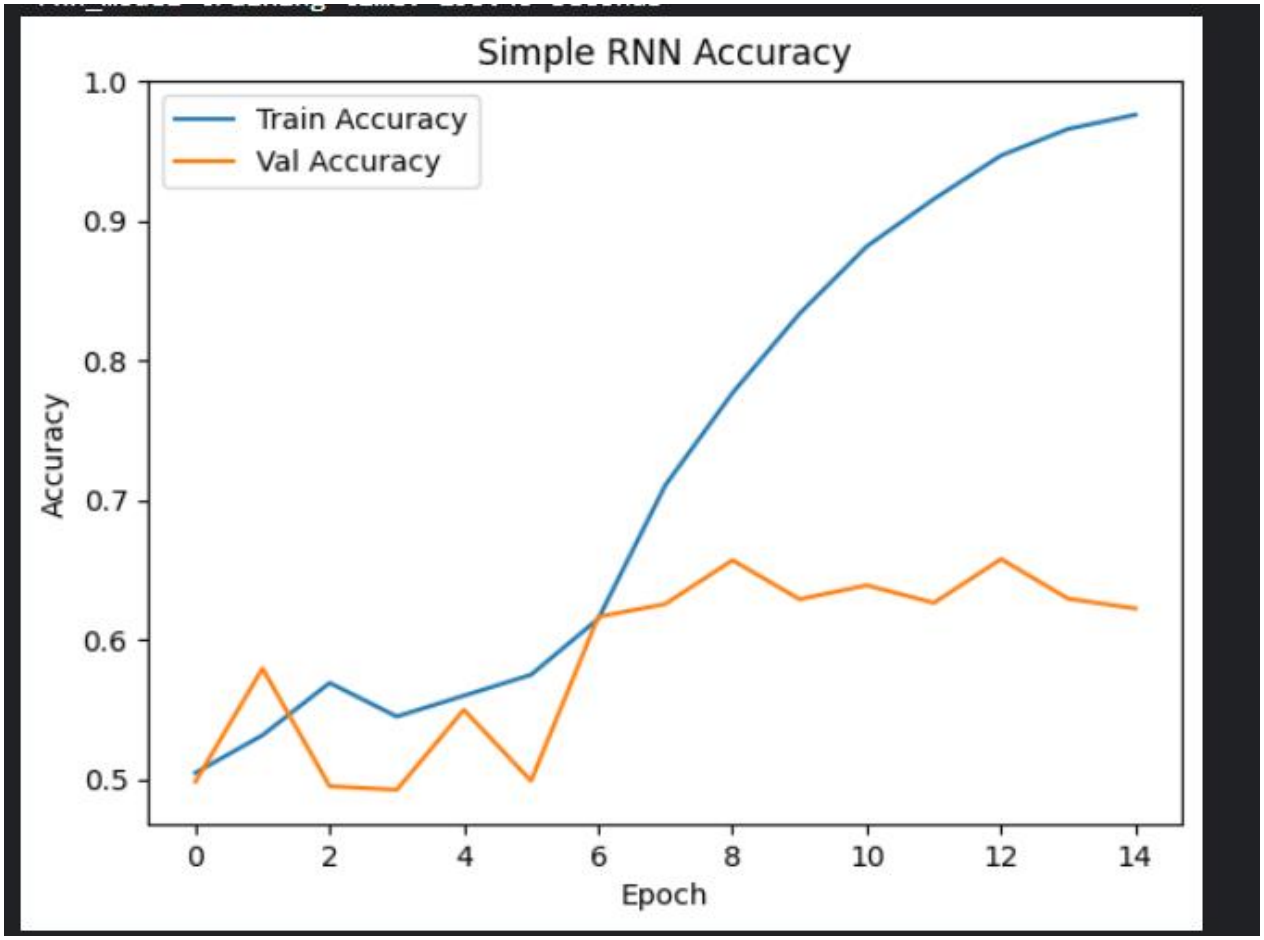

Figure 1: Simple RNN Accuracy

We can illustrate that training accuracy improves steadily, whereas validation accuracy

fluctuates and plateaus around 0.65. This gap indicates potential overfitting, as the model performance is better on training data than on validation data.

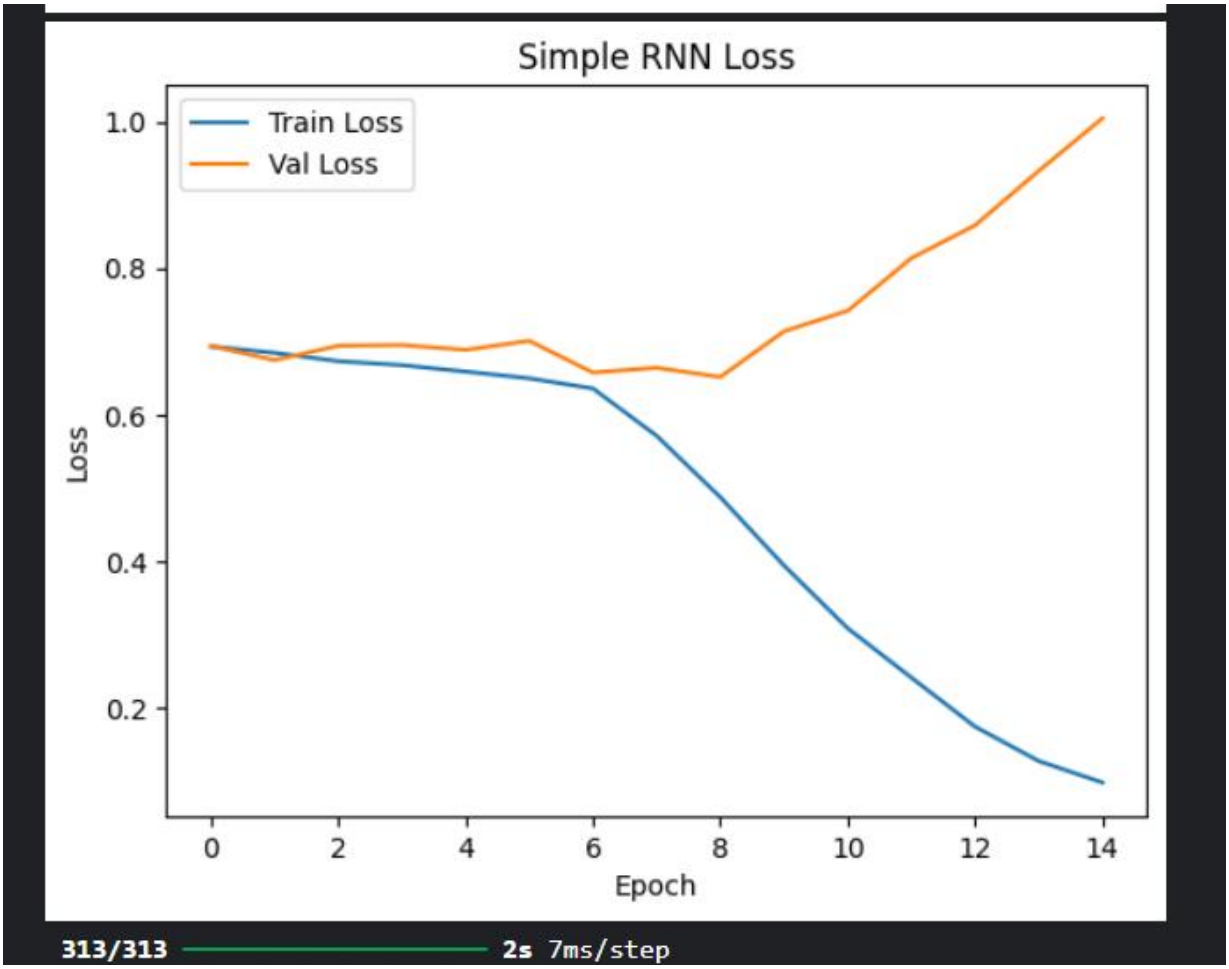

Figure 2: Simple RNN Loss

We can illustrate that a sharp deficit in training loss is observed, but validation loss starts to increase after a certain point, reinforcing signs of overfitting and model instability during generalization.

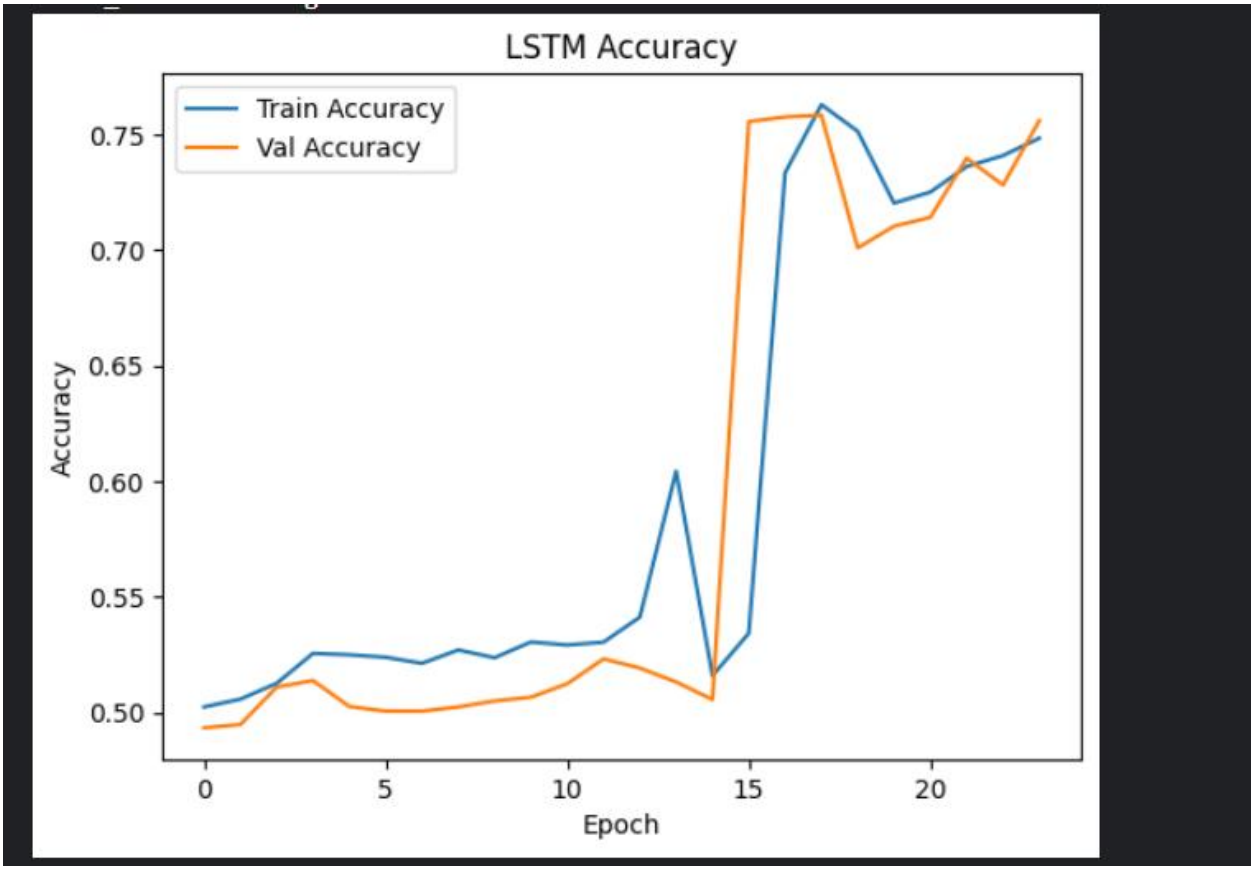

Figure 3: LSTM Accuracy

We can illustrate that, both training and validation accuracy show sharp improvements around the midpoint of training, with validation accuracy closely tracking training accuracy. This indicates strong learning dynamics and better generalization than RNN.

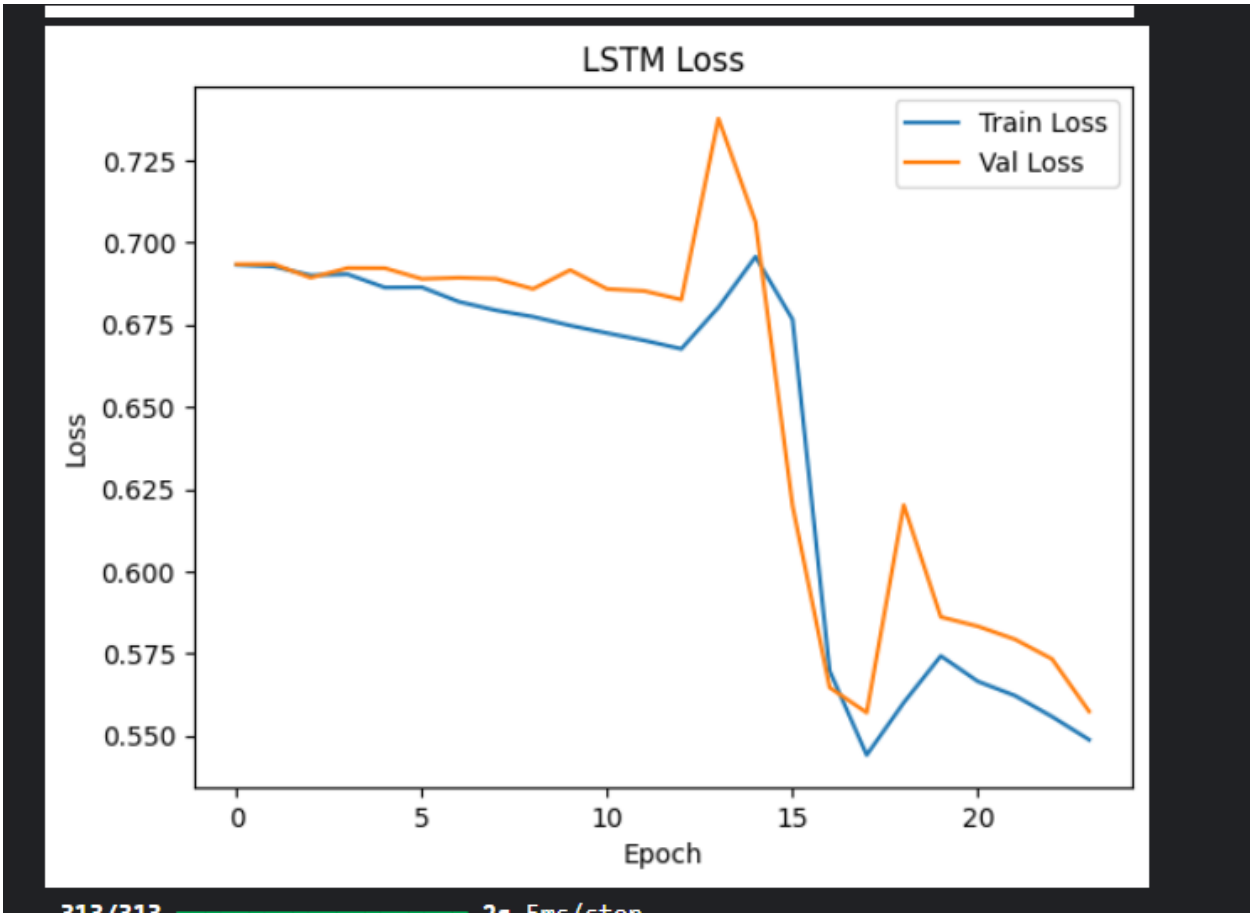

Figure 4: LSTM loss

Training and validation loss decline steadily, with the curves aligning closely post-epoch 15. This alignment supports the observed improvements in accuracy and model stability.

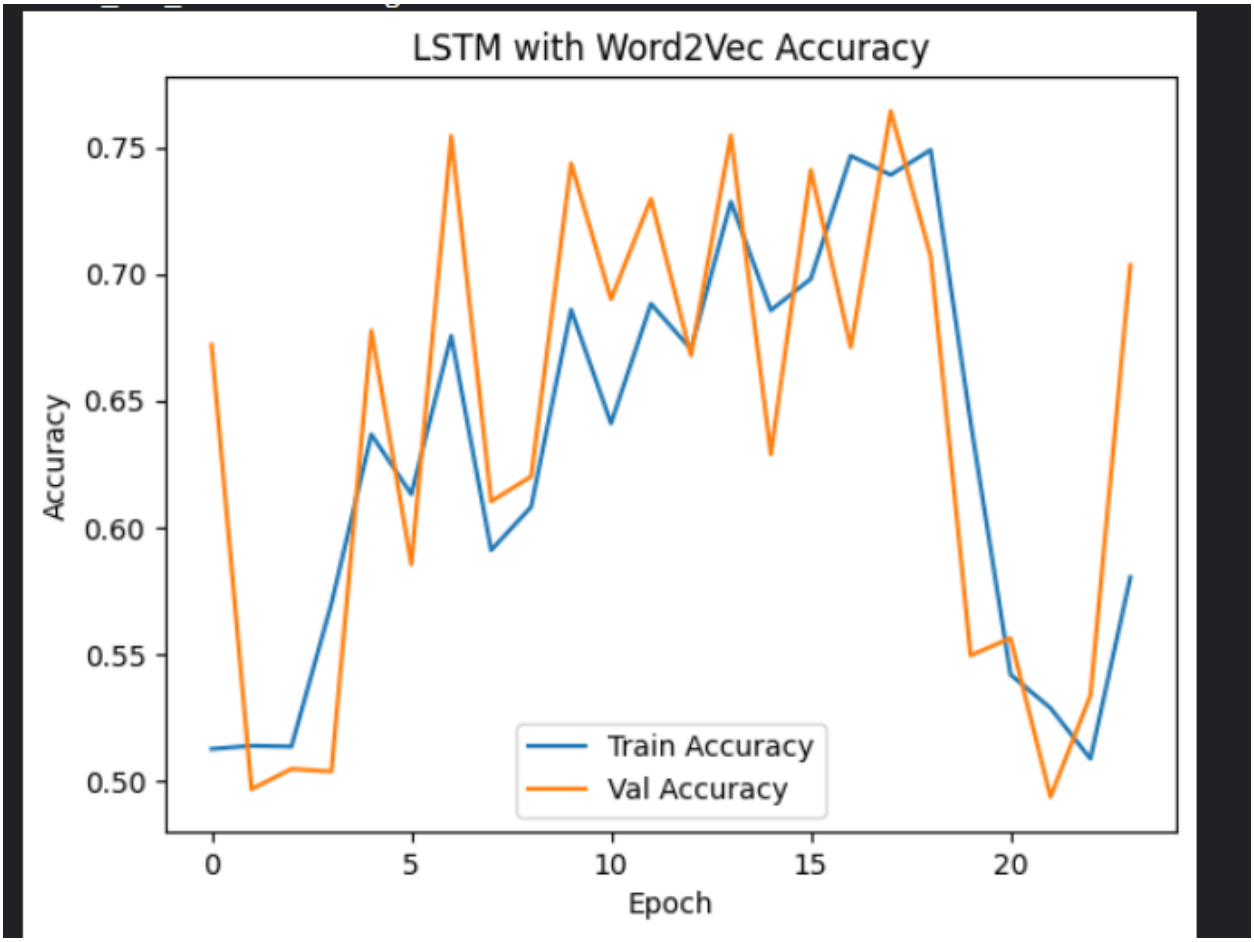

Figure 5: LSTM with Word2Vec Accuracy

Accuracy trends for both training and validation sets are volatile. This fluctuation might result from frozen embedding weights or variability in pretrained vector alignment, suggesting the model's sensitivity to initial embedding representations.

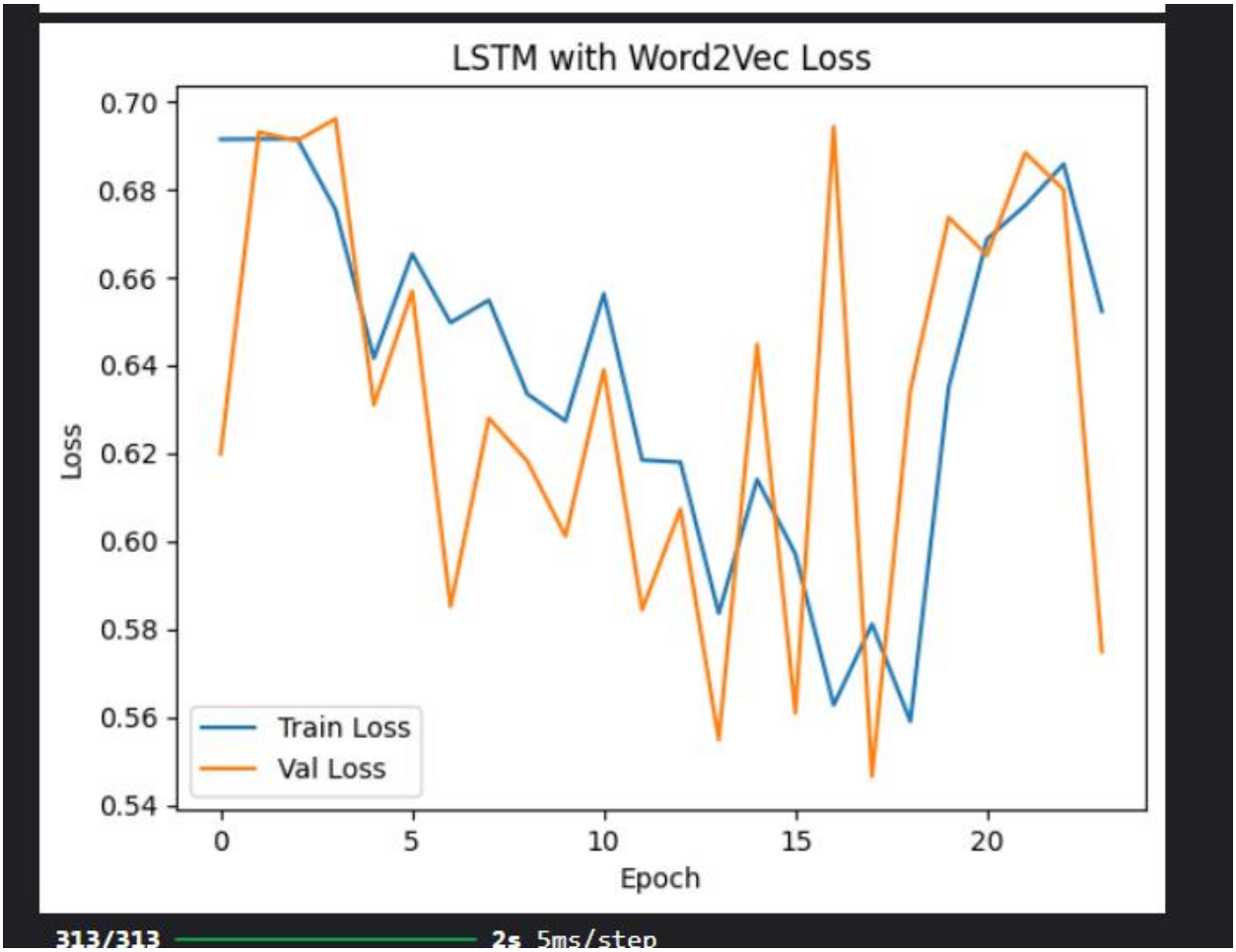

Figure 6: LSTM with Word2Vec Loss

We can illustrates that loss curves fluctuate considerably for training and validation data both. Although the final losses are relatively low, the instability throughout training indicates a need for better regularization or tuning.

4.6. Main Insights from Visualizations

1. The Simple RNN model shows a steady surge in training accuracy but plateaus during validation accuracy early, suggesting underfitting.

2. The LSTM model exhibits a sharp improvement in both training and validation accuracy midway through training, demonstrating a successful learning pattern.

3. LSTM with Word2Vec reveals fluctuating trends, especially in validation accuracy and loss, which may suggest some overfitting or model sensitivity to initialization or embedding variance.

4.7. Confusion Matrix Visualization

To further evaluate the complexity of the models, I visualized confusion matrices:

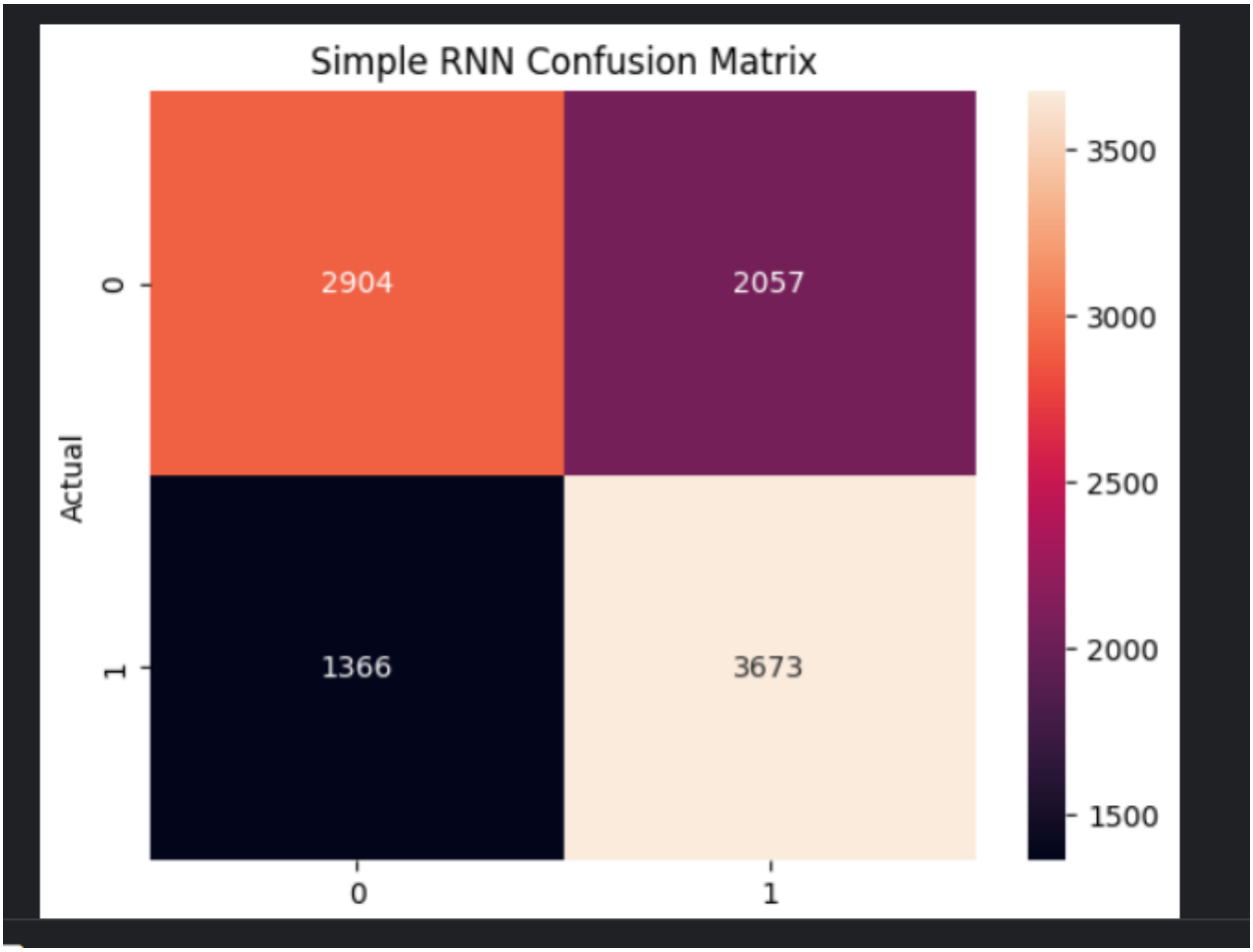

Figure 7: Simple RNN Confusion Matrix

This matrix shows the model correctly classified 2,904 negative and 3,673 positive reviews. However, it misclassified a significant number of negative reviews as positive (2,057), highlighting a recall issue for the negative class.

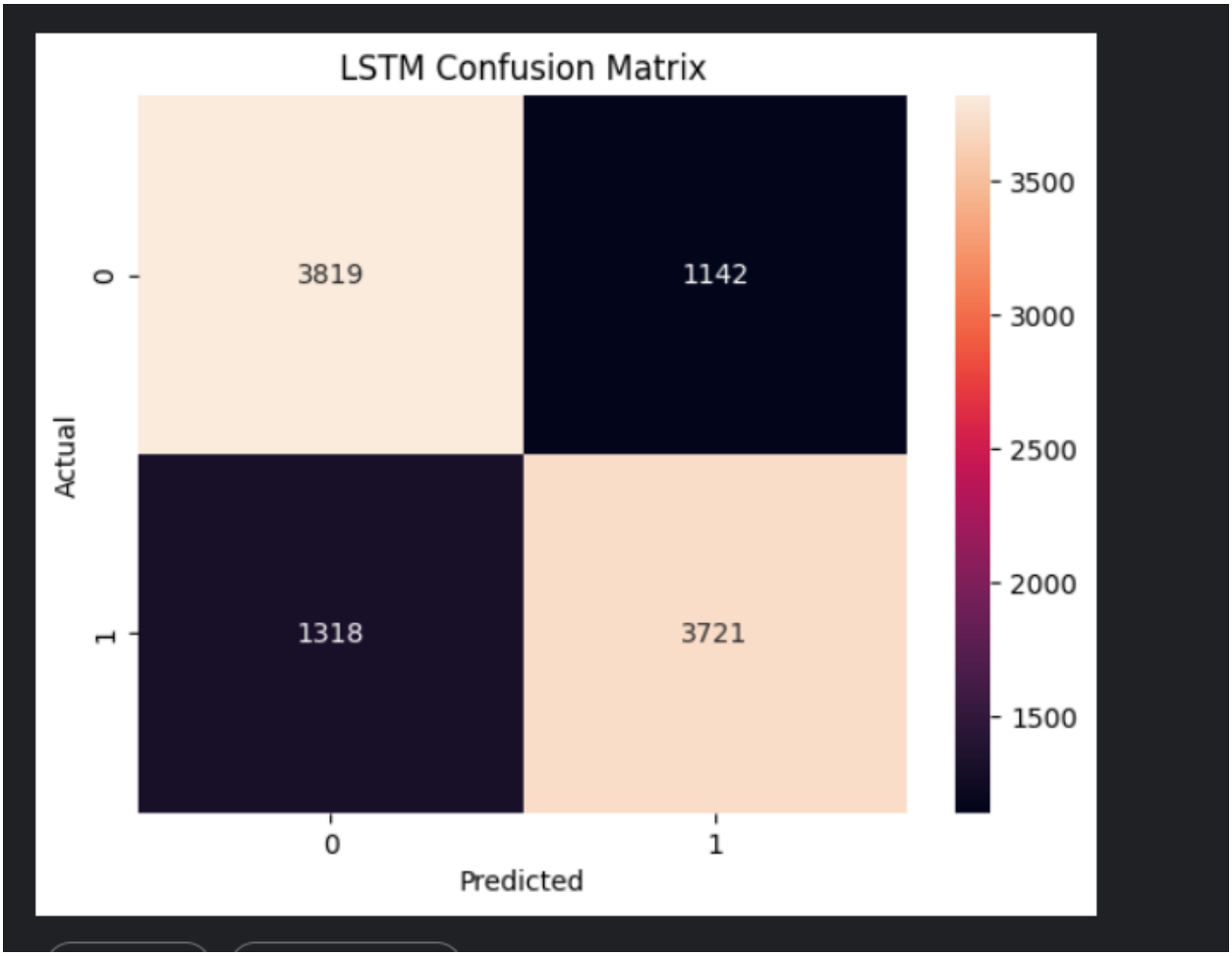

Figure 8: Simple LSTM Confusion Matrix

The LSTM model shows better balance, with 3,819 TN and 3,721 TP. The number of FP and FN has dropped considerably compared to the RNN model.

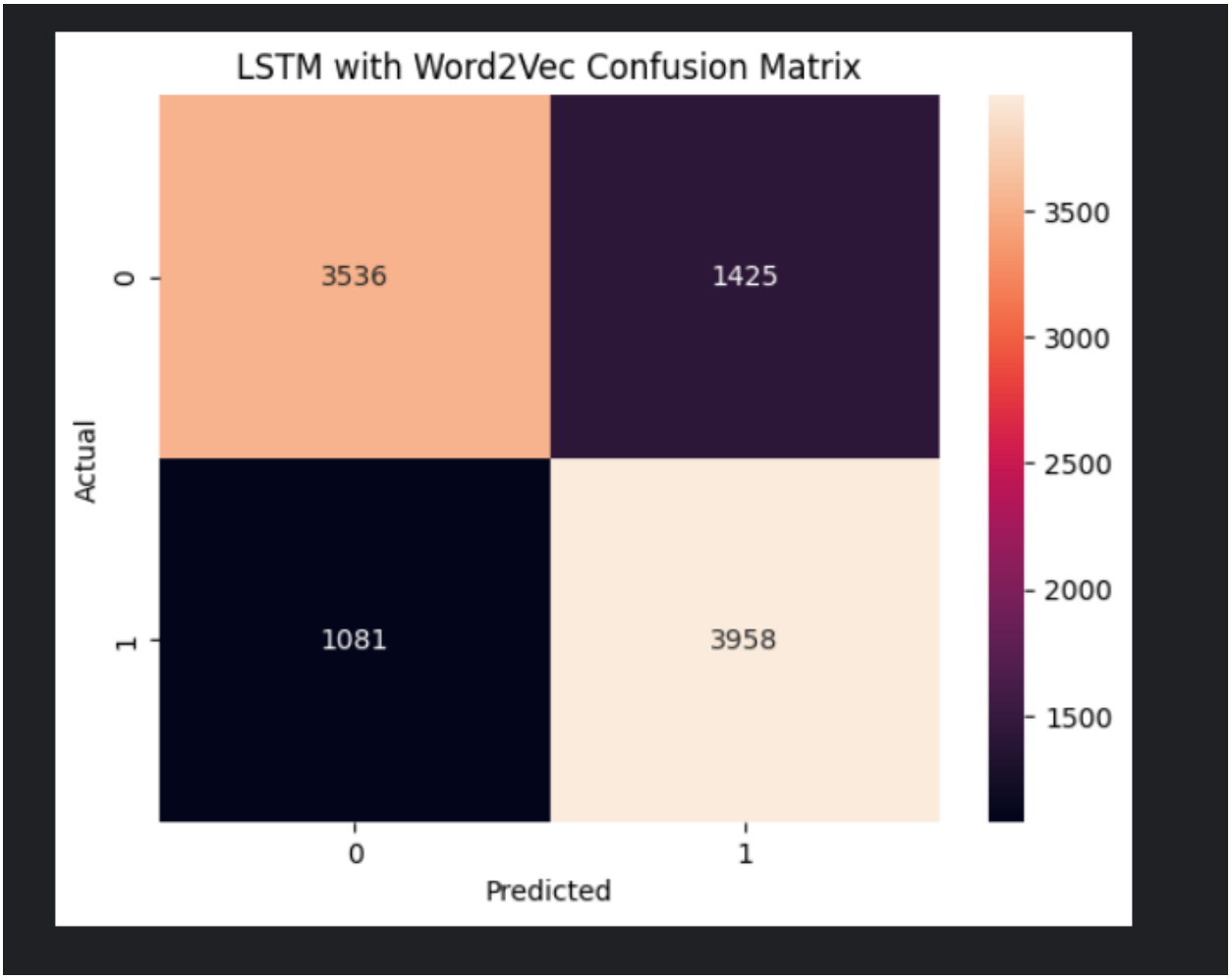

Figure 9: LSTM with Word2Vec Confusion Matrix

This model performs best overall, correctly identifying 3,536 negative and 3,958 positive reviews. False positives and negatives are at their lowest, showcasing strong class separation and generalization capability.

6.7 Summary of Evaluation Metrics

Model	Accuracy	Precision (0/1)	Recall (0/1)	F1-Score (0/1)
Simple RNN	66%	0.68 / 0.64	0.59 / 0.73	0.63 / 0.68
LSTM	75%	0.74 / 0.77	0.77 / 0.74	0.76 / 0.75
LSTM + Word2Vec	75%	0.77 / 0.74	0.71 / 0.79	0.74 / 0.76

Figure 10: Accuracy table

This table complements the visual and textual evaluation by providing a comparative snapshot of how each model performs across multiple classification metrics. It clearly shows the progression of improvement from a basic Simple RNN to more sophisticated LSTM-based models. The LSTM model demonstrates superior recall and F1-score, especially in capturing two of the positive and negative sentiment correctly. The addition of pretrained Word2Vec embeddings offers further refinement in classification quality, particularly improving the recall for positive sentiment. These metrics reinforce the visual feedback provided earlier and confirm that more advanced architectures with semantic-rich embeddings are more effective for sentiment analysis tasks.

5. Conclusion and Future Work

By far into this project, I demonstrate that DL models particularly LSTM and LSTM integrated with Word2Vec embeddings offer substantial improvements over the simpler RNN architecture in sentiment classification. I observe that the LSTM model achieves higher accuracy and better generalization, especially in detecting nuanced sentiment patterns from movie reviews. Adding Word2Vec further improves the model's understanding of semantic relationships among words, even though it occasionally introduces minor variability during training.

Throughout my experimentation, I identify a few challenges. The RNN model struggles to learn long-term dependencies and displays clear underperformance. Although the LSTM and LSTM+Word2Vec models are more effective, I notice some fluctuations in

validation loss and accuracy, which could be due to non-trainable embedding weights or limited data diversity.

Looking ahead, I plan to optimize model performance further through systematic hyperparameter tuning, dropout regularization, and experimenting with larger or more balanced datasets. I am also interested in extending this work using advanced architecture like BERT and attention mechanisms. Lastly, I aim to refine and expand the Stream lit based user interface to support batch predictions, model explanations, and wider usability across different input formats.

6. Appendix

In this section, I describe how I develop, compile, train, evaluate, and visualize my sentiment analysis models. I perform all implementation using Python, TensorFlow/Keras, and Gensim, within the Kaggle notebook environment.

1. Building the Models

I define three separate functions to build the models:

build_rnn_model() creates a Simple RNN model

```
def build_rnn_model():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=128, input_length=max_len),
        SimpleRNN(64),
        Dense(1, activation='sigmoid')
    ])
    return model
```

build_lstm_model() constructs a standard LSTM network.

```
def build_lstm_model():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=128, input_length=max_len),
        LSTM(64),
        Dense(1, activation='sigmoid')
    ])
    return model
```

build_lstm_w2v_model(tokenizer) builds an LSTM model using pretrained Word2Vec embeddings. I use Gensim to load the embeddings and align them with my tokenizer's vocabulary.

```
from gensim.models import KeyedVectors

def build_lstm_w2v_model(tokenizer):
    word2vec_model = KeyedVectors.load_word2vec_format('/kaggle/input/googlenewsvectors/GoogleNews-vectors-negative300.bin', binary=True)
    vocab_size = len(tokenizer.word_index) + 1
    embedding_dim = 300
    embedding_matrix = np.zeros((vocab_size, embedding_dim))
    for word, index in tokenizer.word_index.items():
        if word in word2vec_model:
            embedding_matrix[index] = word2vec_model[word]
    model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=embedding_dim, weights=[embedding_matrix], input_length=max_len, trainable=False),
        LSTM(64),
        Dense(1, activation='sigmoid')
```

2. Compiling the Model

```
def compile_model(model):
    model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
```

Using the compile_model() function, I compile all three models with binary cross-entropy loss, the Adam optimizer, and accuracy as the evaluation metric.

3. Training the Models

```
def train_model(model, name, X_train, y_train, X_val, y_val):
    checkpoint = ModelCheckpoint(f"{name}_best.keras", monitor='val_accuracy', save_best_only=True, mode='max', verbose=0)
    early_stop = EarlyStopping(monitor='val_loss', patience=6, restore_best_weights=True, verbose=0)
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, batch_size=64, callbacks=[checkpoint, early_stop])
    return history
```

I train the models using the train_model() function, which applies early stopping and model checkpointing to preserve the best weights. The function timed_training() wraps this to record and print training time.

4. Visualizing Training Progress

```
def plot_history(history, name):
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title(f'{name} Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title(f'{name} Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
```

The plot_history() function plots the training and validation accuracy/loss over epochs. This helps identify overfitting, underfitting, and convergence behavior.

5. Evaluating Model Performance

```
def evaluate_model(model, name, X_test, y_test):
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    print(f"{name} Classification Report:")
    print(classification_report(y_test, y_pred))
    sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d')
    plt.title(f'{name} Confusion Matrix')
    plt.xlabel("Predicted")
```

Finally, the evaluate_model() function generates a classification report and confusion matrix. It compares predicted vs. actual labels to provide detailed performance feedback.

6. Execution Flow

I execute these functions in sequence for each model:

I first build and compile the model.

I train it using the training and validation sets.

I then visualize the learning curves and evaluate the model on the test set.

```
rnn_model = build_rnn_model()
compile_model(rnn_model)
history_rnn, time_rnn = timed_training(rnn_model, "rnn_model", X_train_pad, y_train, X_val_pad, y_val)
plot_history(history_rnn, "Simple RNN")
evaluate_model(rnn_model, "Simple RNN", X_test_pad, y_test)

lstm_model = build_lstm_model()
compile_model(lstm_model)
history_lstm, time_lstm = timed_training(lstm_model, "lstm_model", X_train_pad, y_train, X_val_pad, y_val)
plot_history(history_lstm, "LSTM")
evaluate_model(lstm_model, "LSTM", X_test_pad, y_test)

lstm_w2v_model = build_lstm_w2v_model(tokenizer)
compile_model(lstm_w2v_model)
history_lstm_w2v, time_lstm_w2v = timed_training(lstm_w2v_model, "lstm_w2v_model", X_train_pad, y_train, X_val_pad, y_val)
plot_history(history_lstm_w2v, "LSTM with Word2Vec")
evaluate_model(lstm_w2v_model, "LSTM with Word2Vec", X_test_pad, y_test)
```

This modular approach ensures clarity, repeatability, and ease of debugging or extension.