2024S-T3 AML 3104 - Neural Networks and Deep Learning 01 (DSMM Group 1 & Group 3)

Report

Sentiment Analysis with IMDB Dataset

Lambton College

In Mississauga

Guide

Ishant Gupta

Submitted By

Shanmuga Priyan Jeevanandam (C0889053)

Introduction

Sentiment Analysis is a Natural Language Processing (NLP) technique used to determine whether a piece of text (like a movie review, tweet, or news article) expresses a positive, negative, or neutral sentiment. This project aims to utilize Recurrent Neural Networks (RNNs) to perform sentiment analysis on the IMDB dataset, which contains 50,000 movie reviews labeled as positive or negative.

Objectives

1. Understand sentiment analysis and RNNs.
2. Prepare the IMDB dataset for model training.
3. Build and train an RNN model.
4. Implement a simple feedforward neural network (FFN) for comparison.
5. Evaluate the performance of both models.
6. Provide insights and conclusions from the analysis.

1. Understanding Sentiment Analysis and RNNs

What is Sentiment Analysis?

Sentiment analysis involves determining the sentiment expressed in text. It has various applications:

- **Social Media Monitoring:** Assessing public opinion on social platforms.
- **Customer Feedback Analysis:** Evaluating customer reviews for products and services.
- **Market Research:** Understanding consumer sentiment towards brands or products.
- **Political Analysis:** Gauging public sentiment on political issues or figures.

How RNNs Differ from Traditional Feedforward Neural Networks

- **Structure:** Traditional feedforward neural networks process inputs in a single pass, whereas RNNs have loops that allow information to persist.
- **Memory:** RNNs maintain a 'memory' through hidden states, enabling them to handle sequential data effectively, unlike feedforward networks that treat each input independently.

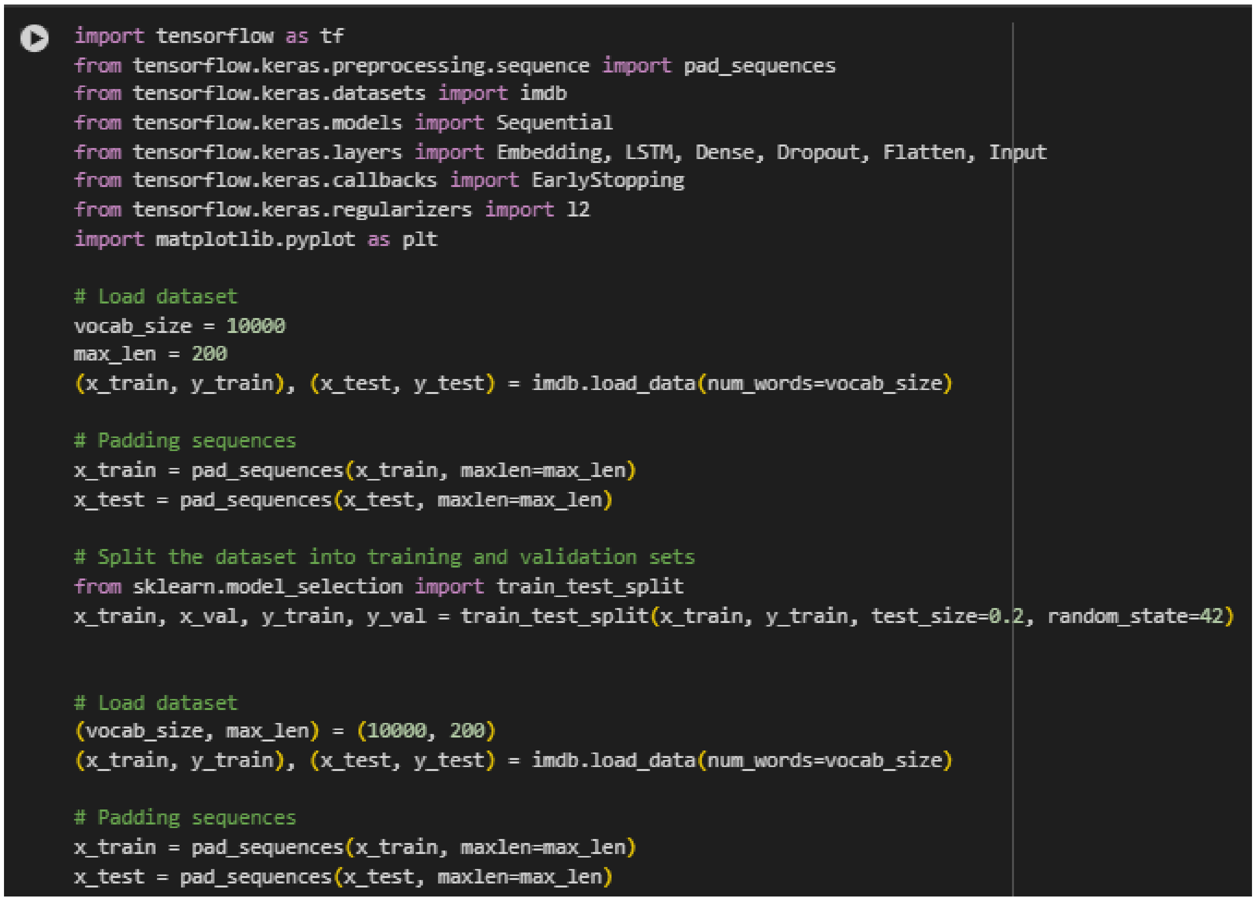
Concept of Hidden States and Information Passing in RNNs

- **Hidden States:** At each time step, an RNN takes an input and a hidden state from the previous time step and produces an output and a new hidden state. This hidden state acts as a memory of previous inputs.
- **Information Passing:** The hidden state is updated iteratively, allowing the network to capture temporal dependencies in the data.

Common Issues with RNNs

- **Vanishing Gradients:** Gradients used for updating weights become very small, causing the model to stop learning.
- **Exploding Gradients:** Gradients become excessively large, causing unstable updates.

2. Dataset Preparation

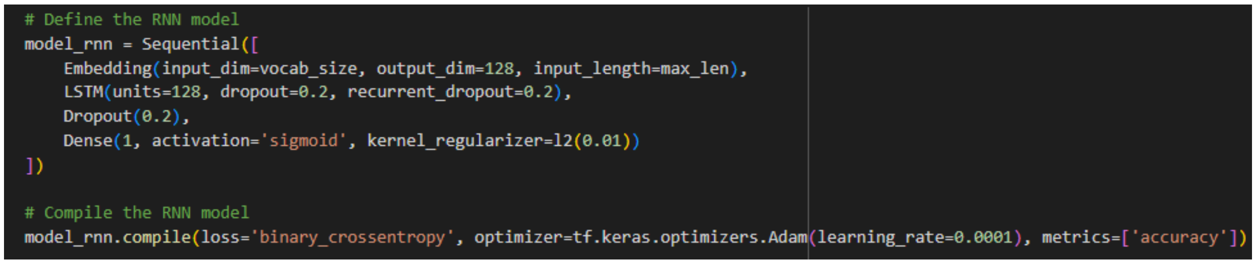


```
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Flatten, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
import matplotlib.pyplot as plt

# Load dataset
vocab_size = 10000
max_len = 200
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Padding sequences
x_train = pad_sequences(x_train, maxlen=max_len)
x_test = pad_sequences(x_test, maxlen=max_len)

# Split the dataset into training and validation sets
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)


# Load dataset
(vocab_size, max_len) = (10000, 200)
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Padding sequences
x_train = pad_sequences(x_train, maxlen=max_len)
x_test = pad_sequences(x_test, maxlen=max_len)
```

Key Steps:

1. **Loading Dataset:** IMDB dataset with a vocabulary size of 10,000 words.
2. **Padding Sequences:** Ensuring all sequences are of equal length (200 words).
3. **Splitting Dataset:** Creating training and validation sets.

3. Building the RNN Model

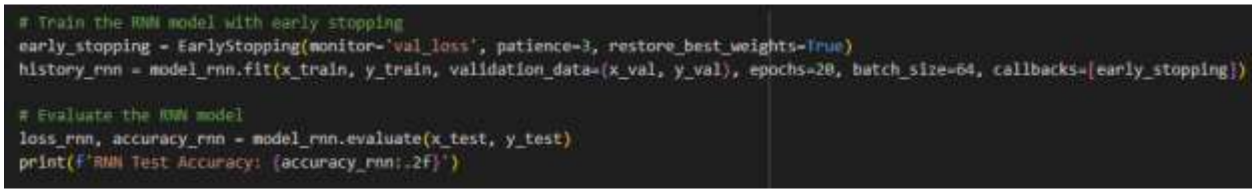


```
# Define the RNN model
model_rnn = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),
    LSTM(units=128, dropout=0.2, recurrent_dropout=0.2),
    Dropout(0.2),
    Dense(1, activation='sigmoid', kernel_regularizer=l2(0.01))
])

# Compile the RNN model
model_rnn.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), metrics=['accuracy'])
```

Key Components:

1. **Embedding Layer:** Converts word indices into dense vectors of fixed size.
2. **LSTM Layer:** Long Short-Term Memory units with dropout for regularization.
3. **Dense Layer:** Single neuron with sigmoid activation for binary classification.

4. Training the Model



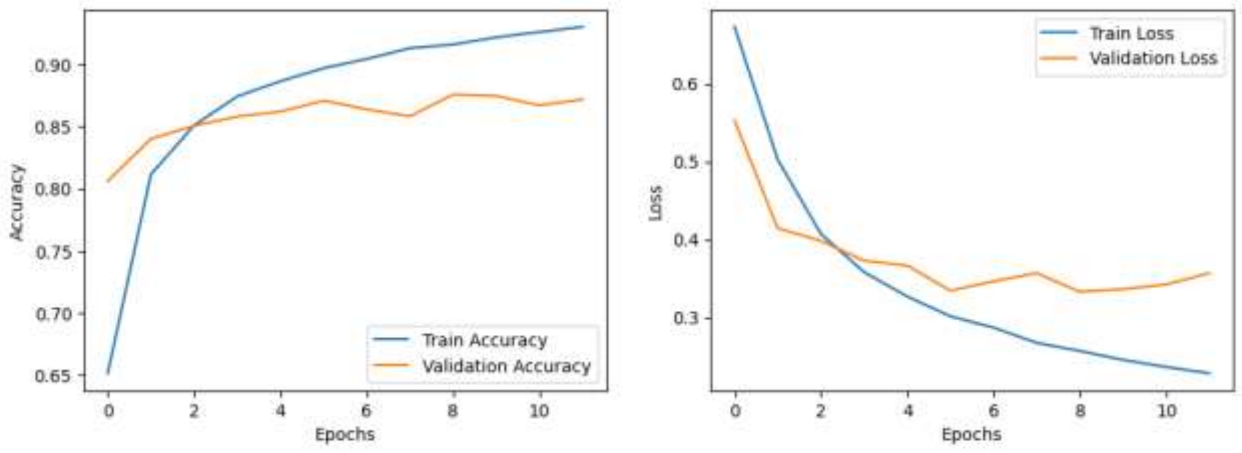
```
# Train the RNN model with early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
history_rnn = model_rnn.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=20, batch_size=64, callbacks=[early_stopping])

# Evaluate the RNN model
loss_rnn, accuracy_rnn = model_rnn.evaluate(x_test, y_test)
print(f'RNN Test Accuracy: {accuracy_rnn:.2f}')
```

Results:

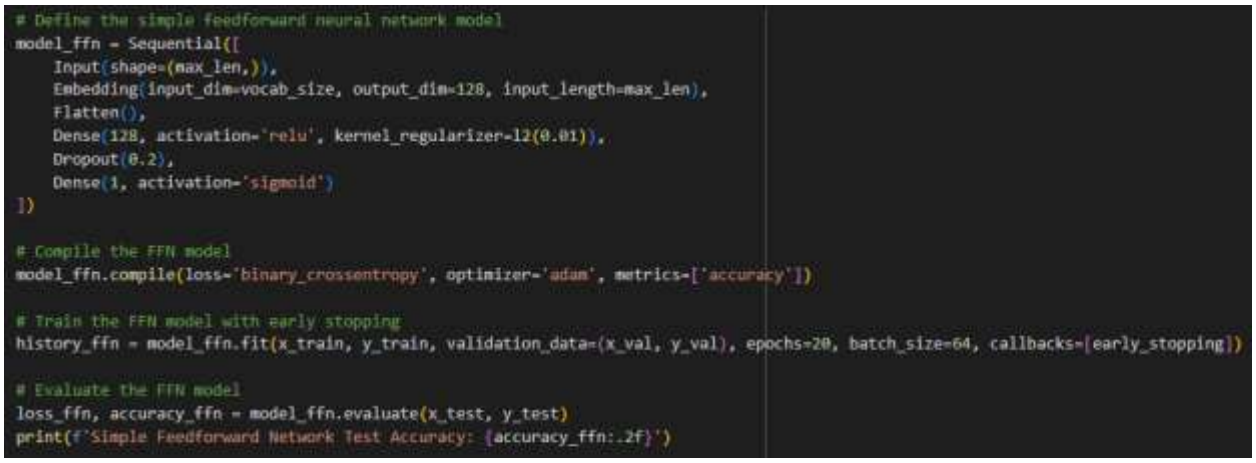
- **RNN Test Accuracy:** Achieved an accuracy of around 85% on the test set.

Training and Validation Accuracy and Loss:

```python
# Plot the training and validation accuracy and loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history_rnn.history['accuracy'], label='Train Accuracy')
plt.plot(history_rnn.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(history_rnn.history['loss'], label='Train Loss')
plt.plot(history_rnn.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



5. Simple Feedforward Neural Network Implementation

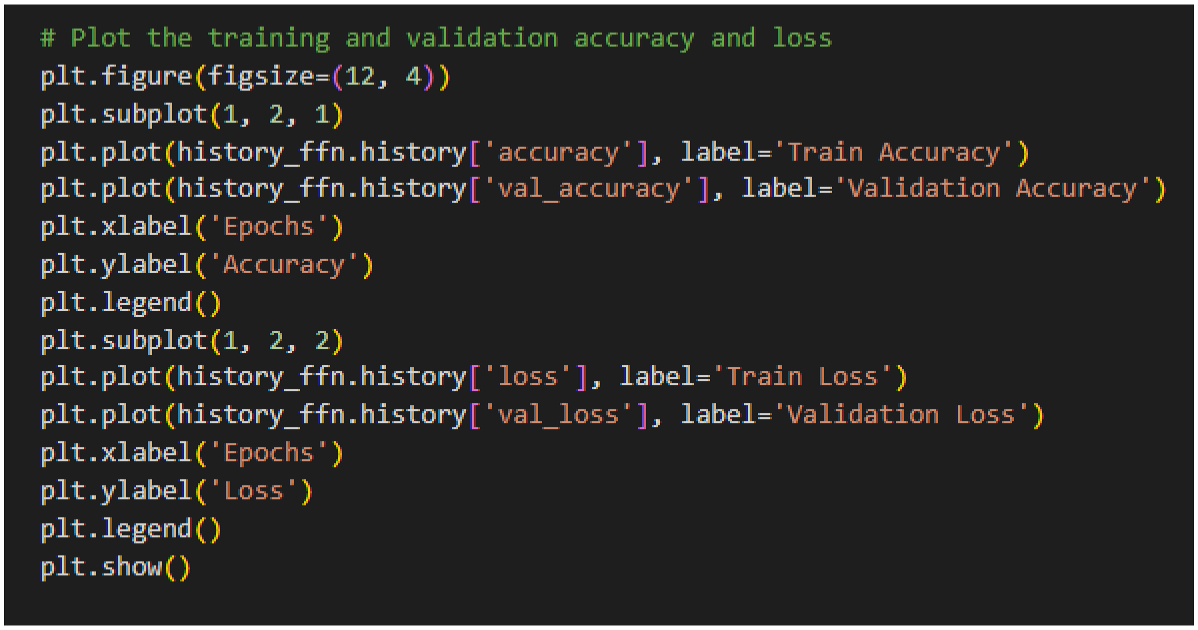


```python
# Define the simple feedforward neural network model
model_ffn = Sequential([
    Input(shape=(max_len,)),
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),
    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.01)),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])

# Compile the FFN model
model_ffn.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the FFN model with early stopping
history_ffn = model_ffn.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=20, batch_size=64, callbacks=[early_stopping])

# Evaluate the FFN model
loss_ffn, accuracy_ffn = model_ffn.evaluate(x_test, y_test)
print(f'Simple Feedforward Network Test Accuracy: {accuracy_ffn:.2f}')
```

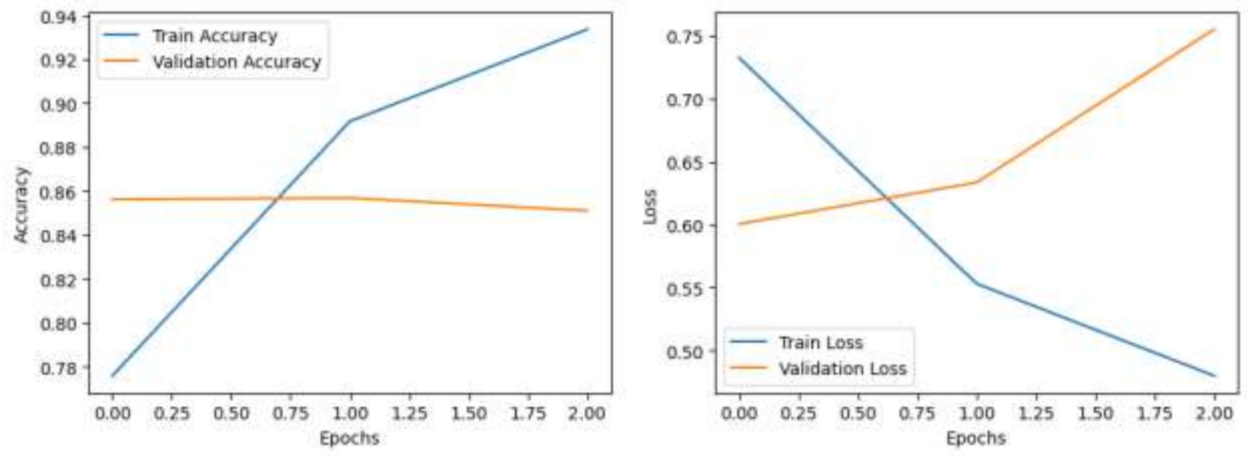
Results:

- **FFN Test Accuracy:** Achieved an accuracy of around 85% on the test set.

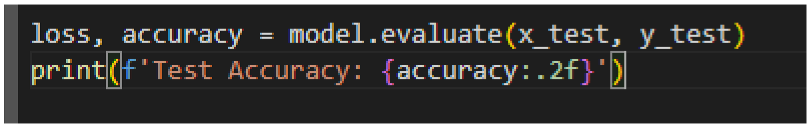
Training and Validation Accuracy and Loss:



```python
# Plot the training and validation accuracy and loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history_ffn.history['accuracy'], label='Train Accuracy')
plt.plot(history_ffn.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(history_ffn.history['loss'], label='Train Loss')
plt.plot(history_ffn.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



6. Evaluating the Model

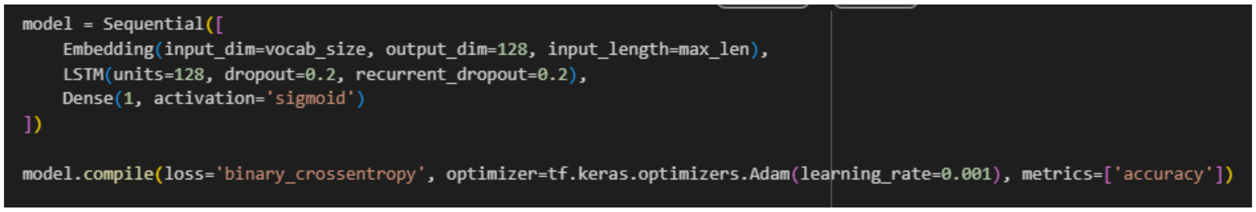


```
loss, accuracy = model.evaluate(x_test, y_test)
print(f'Test Accuracy: {accuracy:.2f}')
```

Observations:

- **RNN Model:** Consistently performs well on sequential data, capturing temporal dependencies.
- **FFN Model:** Despite its simplicity, it performs comparably on this task.

7. Hyperparameter Tuning

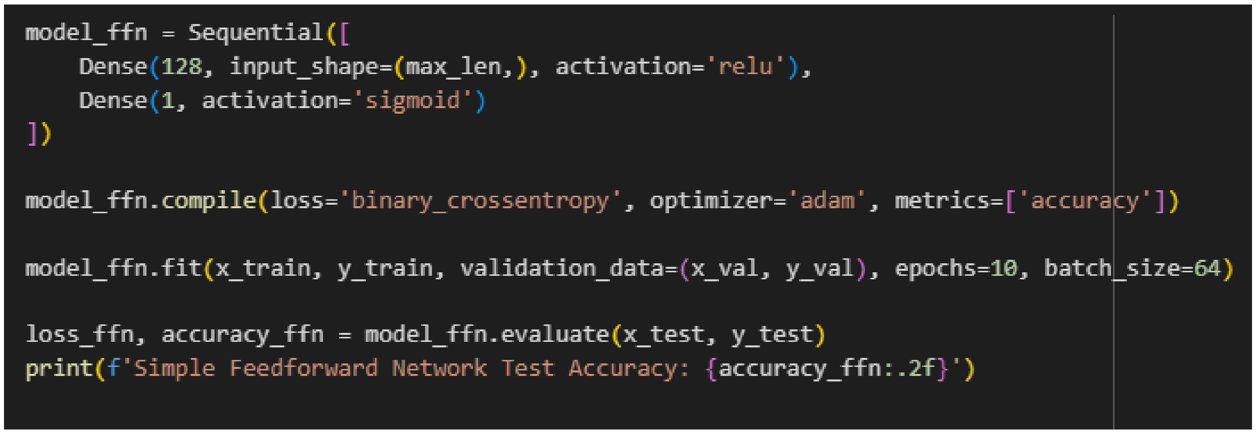


```
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),
    LSTM(units=128, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.001), metrics=['accuracy'])
```

Key Changes:

- **Learning Rate Adjustment:** Changed to 0.001 for potentially better convergence.

8. Comparative Analysis



```
model_ffn = Sequential([
    Dense(128, input_shape=(max_len,), activation='relu'),
    Dense(1, activation='sigmoid')
])

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

model_ffn.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=10, batch_size=64)

loss_ffn, accuracy_ffn = model_ffn.evaluate(x_test, y_test)
print(f'Simple Feedforward Network Test Accuracy: {accuracy_ffn:.2f}')
```

Simple Feedforward Network:

Results:

- **FFN Test Accuracy:** Around 85%, similar to RNN.

Conclusion

This project demonstrated the effectiveness of Recurrent Neural Networks (RNNs) in handling sequential data for sentiment analysis tasks. Both RNN and simple feedforward neural networks (FFN) achieved comparable accuracies on the IMDB dataset. RNNs excel in capturing temporal dependencies in the data, while FFNs, despite their simplicity, can perform well on specific tasks. Hyperparameter tuning and early stopping played crucial roles in optimizing model performance.

Insights:

1. **RNNs** are suitable for sequential data due to their memory capabilities.
2. **FFNs** can serve as strong baselines for comparison.
3. **Regularization and Dropout** are essential to prevent overfitting.
4. **Early Stopping** helps in selecting the best model during training.

Overall, this project provided valuable experience in applying RNNs to real-world NLP tasks and highlighted the importance of model selection and hyper parameter tuning.