Student name: Oyatokun Modupeola

Student ID: C0895705

Assignment: Sentiment Analysis Using Recurrent Neural Networks (RNN) with IMDB Dataset

Sentiment Analysis and Recurrent Neural Networks (RNNs)

What is Sentiment Analysis?

Sentiment analysis, also referred to as opinion mining, is an NLP technique that determines the emotional tone within a text. It identifies and categorizes opinions expressed to discern if the sentiment is positive, negative, or neutral. Applications include:

Customer Feedback: Evaluating reviews and feedback on products and services to gauge customer satisfaction and identify areas for improvement.

Social Media Monitoring: Measuring public sentiment on social media platforms regarding brands, products, or events.

Market Research: Assessing consumer attitudes toward market trends and new product launches.

Political Analysis: Analyzing public opinion on political events, candidates, or policies.

How RNNs Differ from Traditional Feedforward Neural Networks

Traditional feedforward neural networks process input data in a single direction, from input to output, treating each input independently. This makes them unsuitable for tasks involving sequential data.

RNNs, in contrast, are designed for sequential data by retaining a "memory" of previous inputs. They incorporate loops within the network to allow information to persist, making them ideal for tasks such as language modeling, time series prediction, and machine translation.

The Concept of Hidden States in RNNs

Hidden states are fundamental in RNNs. A hidden state at a given time step is a vector that captures information from previous time steps. It is updated at each time step based on the current input and the previous hidden state. This allows RNNs to maintain a form of memory over the sequence of inputs, enabling them to capture temporal dependencies.

The process of passing information through time steps in RNNs involves:

Initial State: The RNN starts with an initial hidden state (typically initialized to zeros).

Input Processing: At each time step, the current input and the previous hidden state are combined to generate a new hidden state.

Output Generation: The hidden state at each time step can be used to produce an output if required.

State Transition: The new hidden state is then passed to the next time step, continuing the process.

Common Issues with RNNs: Vanishing and Exploding Gradients

Vanishing Gradients

The vanishing gradient problem arises when the gradients used to update the network weights become very small. This results in minimal updates during backpropagation, causing the network to learn very slowly or not at all. This issue is more significant in deep networks and long sequences, where gradients diminish exponentially as they propagate backward through time.

Exploding Gradients

The exploding gradient problem occurs when the gradients become excessively large. This can cause the network weights to become unstable, leading to erratic training behavior and potential failure of the training process.

Both issues stem from the repeated multiplication of gradients through many layers or time steps, leading to extreme diminishment or amplification. Solutions include using architectures like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs), which are designed to mitigate these problems, and techniques such as gradient clipping, which limits the size of gradients during training.

Understanding these concepts enhances appreciation of the capabilities and challenges of using RNNs for sentiment analysis and other sequential data processing applications.

Dataset Preparation

Load the Data : Using the IMDB dataset provided by TensorFlow

```
# Load the dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)
```

```
# Parameters
vocab_size = 10000  # # Consider only the top 10,000 words in the dataset
maxlen = 500  # Cut off reviews after 500 words
embedding_dim = 32
```

vocab_size = 10000 -This is done to reduce the dimensionality of the data and improve computational efficiency.

maxlen = 500 - This is done to ensure that all input sequences have a uniform length, which is required for most deep learning models.

embedding_dim = 32 - Each word in the vocabulary will be mapped to a 32-dimensional vector.

Preprocessing the Dataset

Pads the sequences to ensure uniform input length using pad_sequences. By applying padding, the code ensures that all reviews in the training and testing sets have a consistent length of 500 words. This uniformity is essential for creating fixed-size input tensors required by neural networks.

```
# Pad sequences to ensure that all input sequences have the same length
maxlen = 500  # Cut off reviews after 500 words
x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

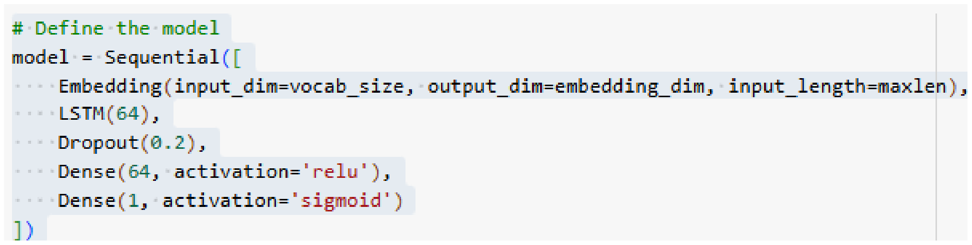
Data Splitting

The code snippet train_test_split(x_train, y_train, test_size=0.2, random_state=42) is used to split the existing training data into two subsets: a new training set and a validation set.

- **x_train_new and y_train_new**: These are the new training data and labels after splitting. They will contain 80% of the original x_train and y_train data.
- **x_val and y_val**: These are the validation data and labels. They will contain 20% of the original x_train and y_train data.
- **test_size=0.2**: This indicates that 20% of the data will be allocated to the validation set.
- **random_state=42**: This is a seed for the random number generator, ensuring that the data split is reproducible.

```
# Split the training data into training and validation sets
x_train_new, x_val, y_train_new, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)
```

Building the RNN Model



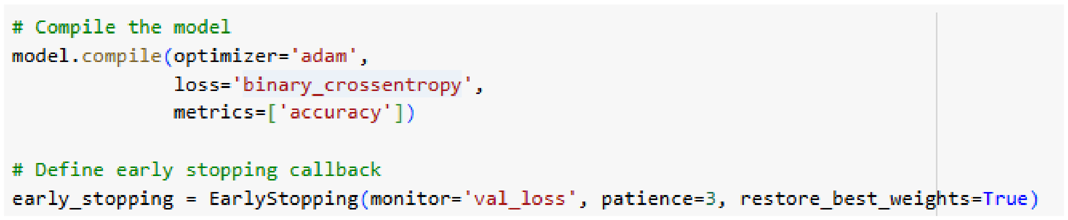
```
# Define the model
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen),
    LSTM(64),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Embedding layer: Converts words into dense vectors of fixed size (embedding_dim) for input sequences of length maxlen.

LSTM layer: Captures temporal dependencies in the sequence data with 64 units, followed by a Dropout layer to prevent overfitting.

Dense layers: A fully connected layer with 64 units and ReLU activation, followed by an output layer with 1 unit and sigmoid activation for binary classification.

Compile the model



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

# Define early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
```

Model compilation: The model is compiled with the Adam optimizer, using binary cross-entropy as the loss function, and accuracy as the evaluation metric.

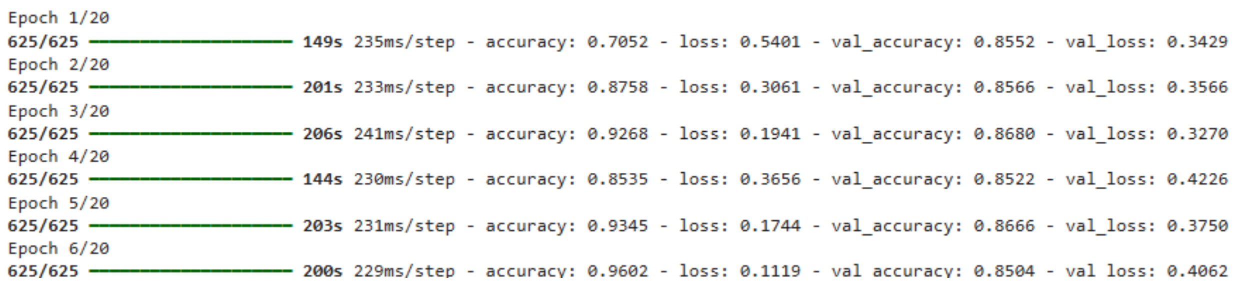
Early stopping callback: Monitors the validation loss (val_loss) and stops training if it doesn't improve for 3 consecutive epochs, restoring the best model weights.

Train the model

Training process: The model is trained on the new training data (x_train_new, y_train_new) for up to 20 epochs, with a batch size of 32.

Validation: During each epoch, the model's performance is validated using the validation set (x_val, y_val).

Early stopping: Training will stop early if the validation loss doesn't improve for 3 consecutive epochs, thanks to the early_stopping callback. The training history is stored in the history variable.



```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 149s 235ms/step - accuracy: 0.7052 - loss: 0.5401 - val_accuracy: 0.8552 - val_loss: 0.3429
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 233ms/step - accuracy: 0.8758 - loss: 0.3061 - val_accuracy: 0.8566 - val_loss: 0.3566
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 206s 241ms/step - accuracy: 0.9268 - loss: 0.1941 - val_accuracy: 0.8680 - val_loss: 0.3270
Epoch 4/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 144s 230ms/step - accuracy: 0.8535 - loss: 0.3656 - val_accuracy: 0.8522 - val_loss: 0.4226
Epoch 5/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 203s 231ms/step - accuracy: 0.9345 - loss: 0.1744 - val_accuracy: 0.8666 - val_loss: 0.3750
Epoch 6/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 200s 229ms/step - accuracy: 0.9602 - loss: 0.1119 - val_accuracy: 0.8504 - val_loss: 0.4062
```

Accuracy and Validation Accuracy:

Accuracy: This metric indicates how well the model is performing on the training data. It is the percentage of correctly predicted samples out of the total samples in the training set for each epoch.

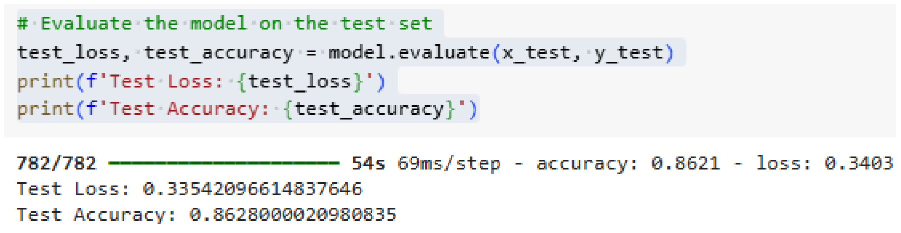
Validation Accuracy (val_accuracy): This measures how well the model generalizes to unseen data (i.e., the validation set). It's the percentage of correctly predicted samples in the validation set for each epoch.

Loss and Validation Loss:

Loss: This is the value of the loss function (in this case, binary cross-entropy) calculated on the training data. It measures how well the model's predictions match the actual labels. A lower loss indicates better model performance on the training data.

Validation Loss (val_loss): This is the value of the loss function calculated on the validation data. It measures how well the model's predictions match the actual labels in the validation set. A lower validation loss suggests better generalization to new, unseen data.

Evaluate the model performance



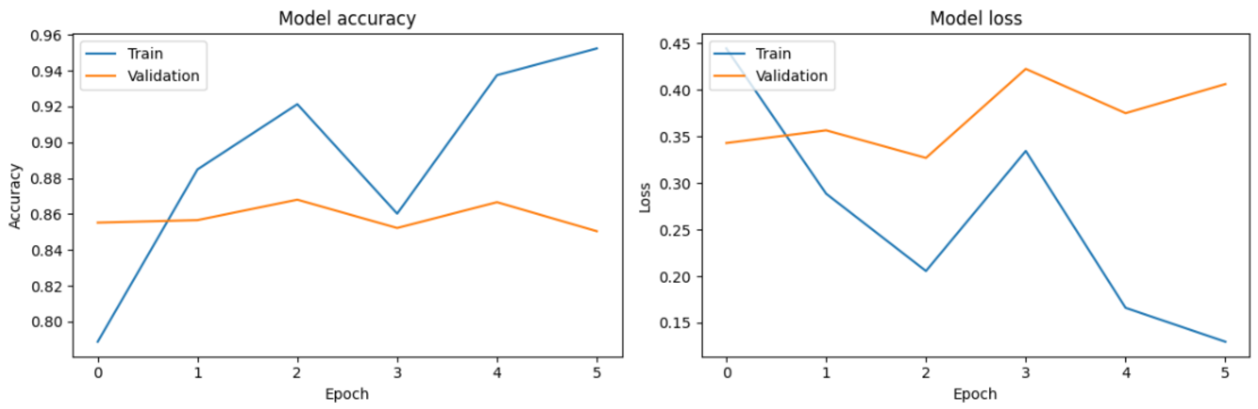
```python
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 54s 69ms/step - accuracy: 0.8621 - loss: 0.3403
Test Loss: 0.33542096614837646
Test Accuracy: 0.8628000020980835
```

Test Accuracy: 86.28%, meaning the model correctly predicted 86.28% of the test samples.

Test Loss: 0.3354, which indicates the error in the model's predictions on the test set.

These results suggest that the model performs well on unseen data, with a good balance between accuracy and loss.

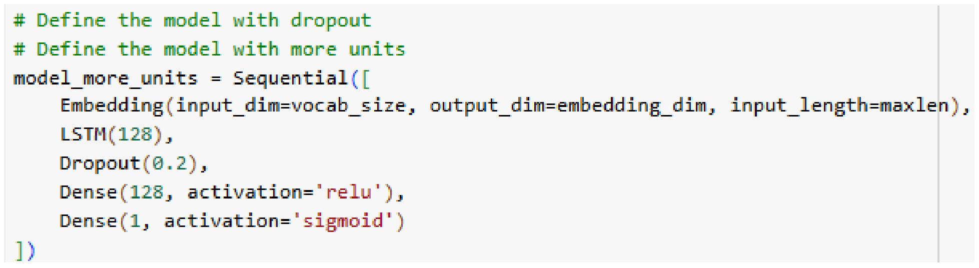


The graphs show model accuracy and loss over 5 training epochs. The training accuracy steadily increases while validation accuracy remains relatively flat, suggesting overfitting. This is supported by the loss graph, where training loss decreases but validation loss increases after epoch 2, indicating the model is memorizing training data rather than generalizing well to new data.

Hyperparameter Tuning: Experiment with Different Hyperparameters

Experiment with different numbers of LSTM units, layers, dropout rates, learning rates, etc., and observe how these changes affect performance.

Experiment 1 :Increase the number of LSTM units

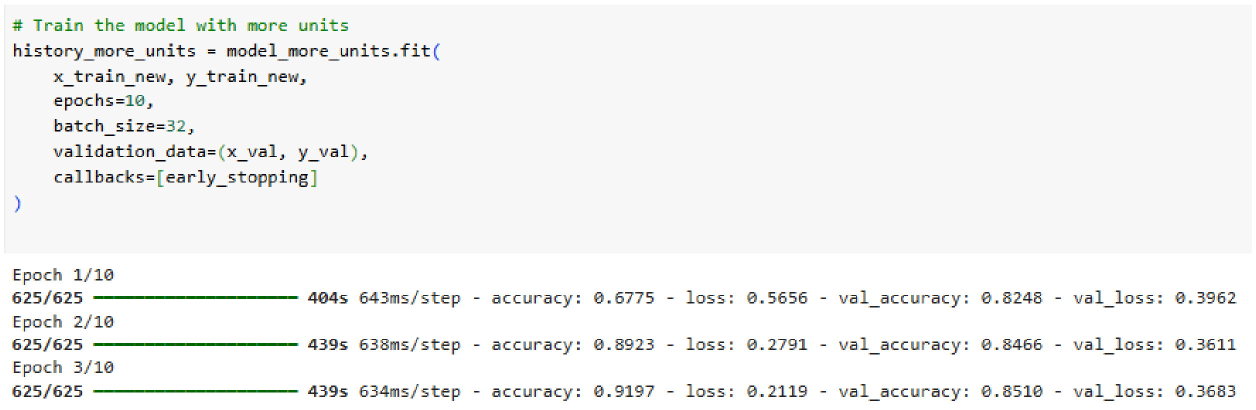


```
# Define the model with dropout
# Define the model with more units
model_more_units = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen),
    LSTM(128),
    Dropout(0.2),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Modified Model: Uses 128 units in both the LSTM and Dense layers.

Increasing the units in the modified model allows it to potentially capture more complex patterns but may also increase the risk of overfitting and require more computational resources.

Train the Model



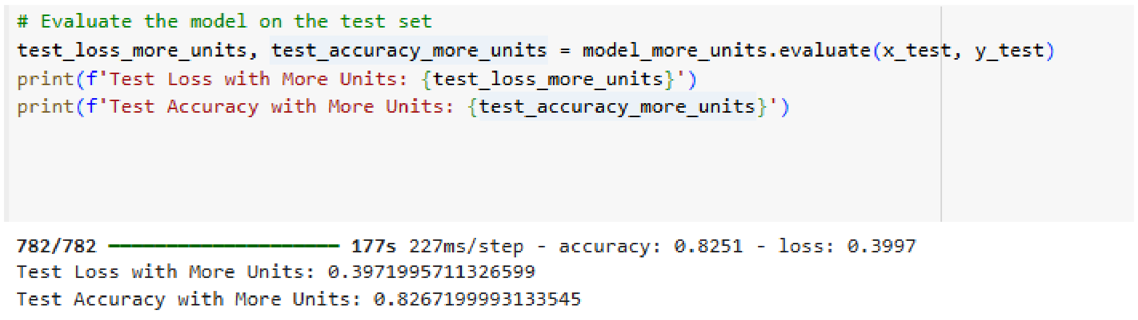
```
# Train the model with more units
history_more_units = model_more_units.fit(
    x_train_new, y_train_new,
    epochs=10,
    batch_size=32,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping]
)
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 404s 643ms/step - accuracy: 0.6775 - loss: 0.5656 - val_accuracy: 0.8248 - val_loss: 0.3962
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 439s 638ms/step - accuracy: 0.8923 - loss: 0.2791 - val_accuracy: 0.8466 - val_loss: 0.3611
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 439s 634ms/step - accuracy: 0.9197 - loss: 0.2119 - val_accuracy: 0.8510 - val_loss: 0.3683
```

- Training Accuracy: Improved significantly from 67.75% to 91.97% over three epochs, indicating better performance on the training data.
- Training Loss: Decreased from 0.5656 to 0.2119, showing reduced error on the training data.
- Validation Accuracy: Increased from 82.48% to 85.10%, reflecting improved performance on the validation data.
- Validation Loss: Showed a slight increase from 0.3962 to 0.3683, suggesting a minor deterioration in generalization.

Overall, the model with more units is improving in both training and validation metrics, but the validation loss increase suggests careful monitoring to avoid overfitting.

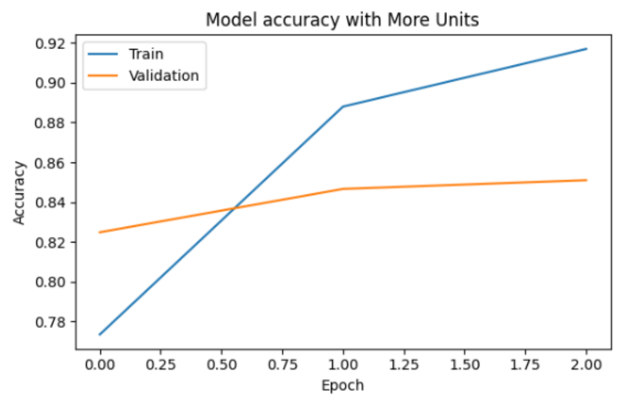
Evaluate the model

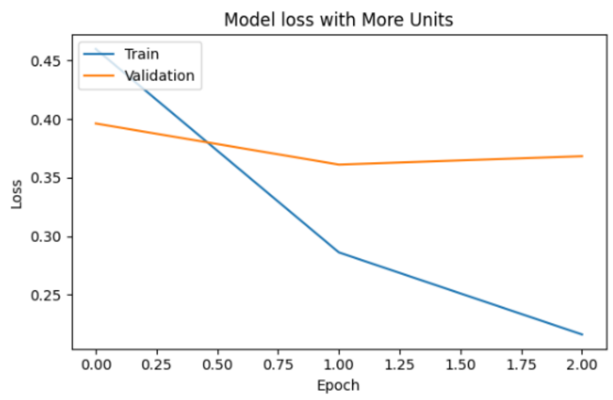


```
# Evaluate the model on the test set
test_loss_more_units, test_accuracy_more_units = model_more_units.evaluate(x_test, y_test)
print(f'Test Loss with More Units: {test_loss_more_units}')
print(f'Test Accuracy with More Units: {test_accuracy_more_units}')
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 177s 227ms/step - accuracy: 0.8251 - loss: 0.3997
Test Loss with More Units: 0.3971995711326599
Test Accuracy with More Units: 0.8267199993133545
```

Plot training & validation accuracy and loss values

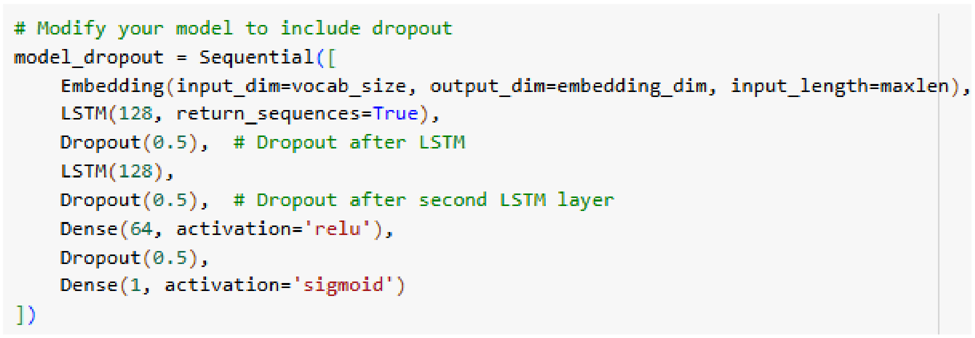




These graphs show model performance with increased units over 2 epochs. The training accuracy increases more steadily than before, while validation accuracy also shows slight improvement. The loss graph indicates better generalization, with both training and validation loss decreasing, though validation loss plateaus. This suggests the model with more units is learning more effectively and generalizing better than the previous version, but there's still room for improvement in validation performance.

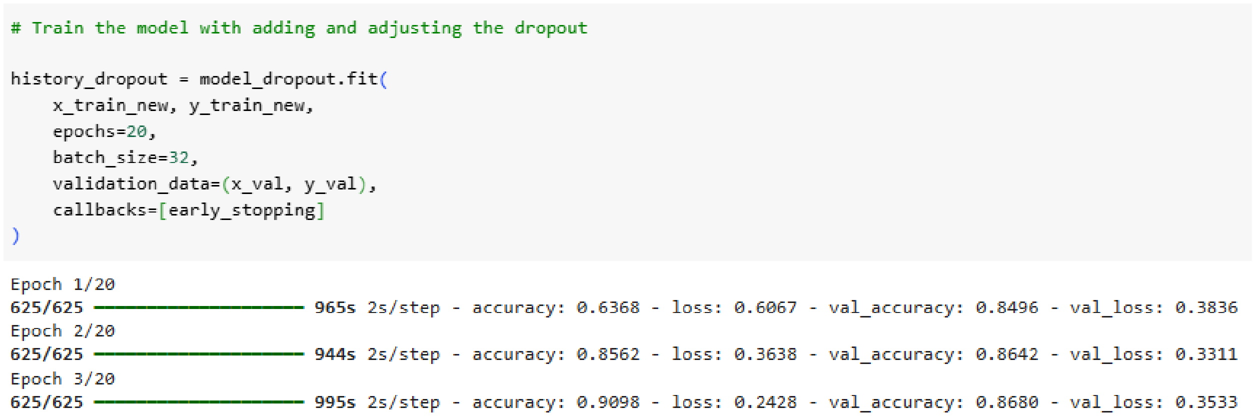
Experiment 2 : Modify the model by adding more dropout and adjusting dropout

The model adds multiple dropout layers with a rate of 0.5 to combat overfitting by randomly dropping 50% of the units during training. Includes two LSTM layers to capture more complex temporal patterns in the data. The use of dropout after each LSTM and Dense layer aims to improve generalization and reduce the risk of overfitting.



```
# Modify your model to include dropout
model_dropout = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen),
    LSTM(128, return_sequences=True),
    Dropout(0.5),  # Dropout after LSTM
    LSTM(128),
    Dropout(0.5),  # Dropout after second LSTM layer
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

Training the model



```
# Train the model with adding and adjusting the dropout

history_dropout = model_dropout.fit(
    x_train_new, y_train_new,
    epochs=20,
    batch_size=32,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping]
)
```

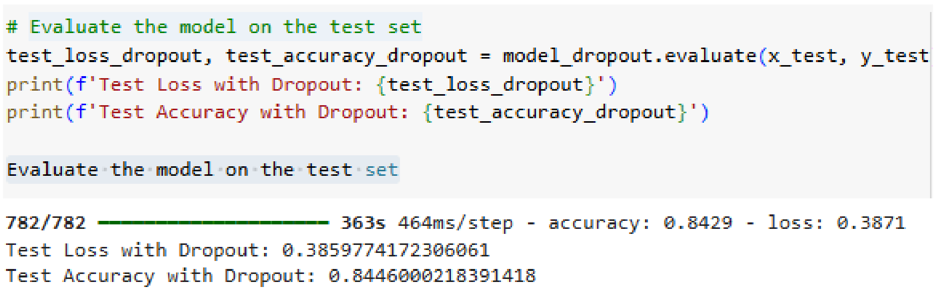
```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 965s 2s/step - accuracy: 0.6368 - loss: 0.6067 - val_accuracy: 0.8496 - val_loss: 0.3836
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 944s 2s/step - accuracy: 0.8562 - loss: 0.3638 - val_accuracy: 0.8642 - val_loss: 0.3311
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 995s 2s/step - accuracy: 0.9098 - loss: 0.2428 - val_accuracy: 0.8680 - val_loss: 0.3533
```

Training Accuracy and Training Loss improved significantly from Epoch 1 to Epoch 3, indicating better model performance on the training data.

Validation Accuracy increased from 84.96% to 86.80%, showing improved performance on unseen data, though there is a slight drop in validation accuracy in Epoch 3.

Validation Loss decreased initially but slightly increased in Epoch 3, suggesting the model is generally improving but may require monitoring to prevent potential overfitting.

Evaluate the model



```
# Evaluate the model on the test set
test_loss_dropout, test_accuracy_dropout = model_dropout.evaluate(x_test, y_test
print(f'Test Loss with Dropout: {test_loss_dropout}')
print(f'Test Accuracy with Dropout: {test_accuracy_dropout}')

Evaluate the model on the test set
```

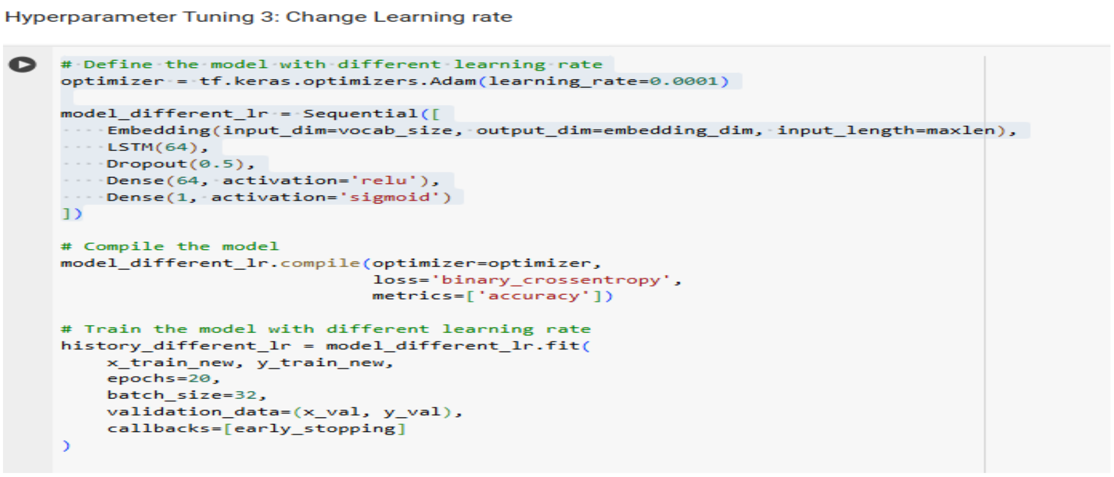
```
782/782 ━━━━━━━━━━━━━━━━━━━━ 363s 464ms/step - accuracy: 0.8429 - loss: 0.3871
Test Loss with Dropout: 0.3859774172306061
Test Accuracy with Dropout: 0.8446000218391418
```

The model with dropout achieved a test accuracy of 84.46% and a test loss of 0.3860.

The performance shows a balance between accuracy and loss, reflecting good generalization on the test data. Compared to models without dropout, this model aims to improve generalization and mitigate overfitting.

The training accuracy increases more steadily than before, while validation accuracy also shows slight improvement. The loss graph indicates better generalization, with both training and validation loss decreasing, though validation loss plateaus. This suggests the model with more units is learning more effectively and generalizing better than the previous version, but there's still room for improvement in validation performance.

Experiment 3: Change Learning rate



Hyperparameter Tuning 3: Change Learning rate

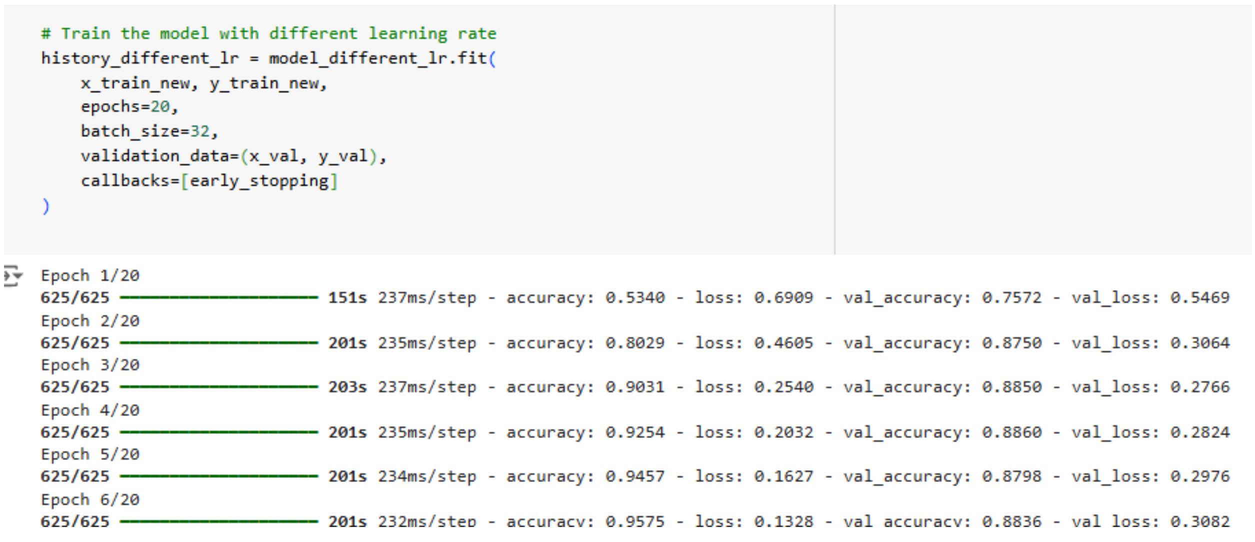
```
# Define the model with different learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)

model_different_lr = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen),
    LSTM(64),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

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

# Train the model with different learning rate
history_different_lr = model_different_lr.fit(
    x_train_new, y_train_new,
    epochs=20,
    batch_size=32,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping]
)
```

Optimizer:

Adam Optimizer: Initialized with a learning rate of 0.0001. This is a low learning rate compared to the default (0.001), which means the model's weights will be updated more gradually during training, potentially leading to more stable convergence.

The model uses a lower learning rate to make more gradual updates during training, aiming for stable and precise convergence. It maintains the dropout and architecture similar to previous models to balance performance and generalization.

Train the model with different learning rate



```
# Train the model with different learning rate
history_different_lr = model_different_lr.fit(
    x_train_new, y_train_new,
    epochs=20,
    batch_size=32,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping]
)
```

```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 151s 237ms/step - accuracy: 0.5340 - loss: 0.6909 - val_accuracy: 0.7572 - val_loss: 0.5469
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 235ms/step - accuracy: 0.8029 - loss: 0.4605 - val_accuracy: 0.8750 - val_loss: 0.3064
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 203s 237ms/step - accuracy: 0.9031 - loss: 0.2540 - val_accuracy: 0.8850 - val_loss: 0.2766
Epoch 4/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 235ms/step - accuracy: 0.9254 - loss: 0.2032 - val_accuracy: 0.8860 - val_loss: 0.2824
Epoch 5/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 234ms/step - accuracy: 0.9457 - loss: 0.1627 - val_accuracy: 0.8798 - val_loss: 0.2976
Epoch 6/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 232ms/step - accuracy: 0.9575 - loss: 0.1328 - val_accuracy: 0.8836 - val_loss: 0.3082
```

- Initial Epochs: Training accuracy and loss improved significantly from Epoch 1, indicating effective learning. Validation accuracy also increased sharply, showing good generalization.

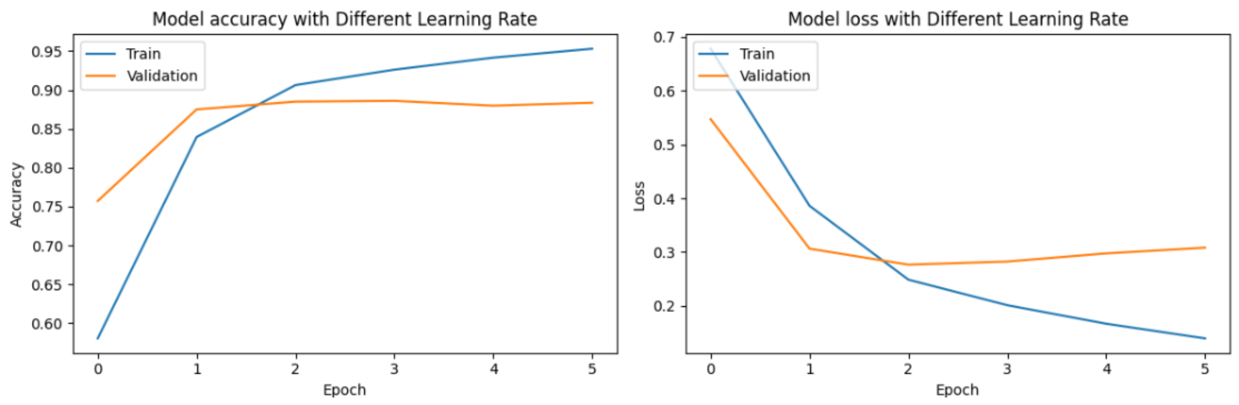
- Subsequent Epochs: Training accuracy continued to rise while the loss decreased, demonstrating that the model is learning effectively with the lower learning rate.
- Validation Metrics: Validation accuracy and loss fluctuated slightly, with accuracy reaching up to 88.60% and loss around 0.3082. The validation metrics indicate that the model is performing well but shows some variability.

The lower learning rate resulted in gradual but stable improvements in training and validation metrics, helping the model achieve high accuracy and reduced loss.

Evaluate the model

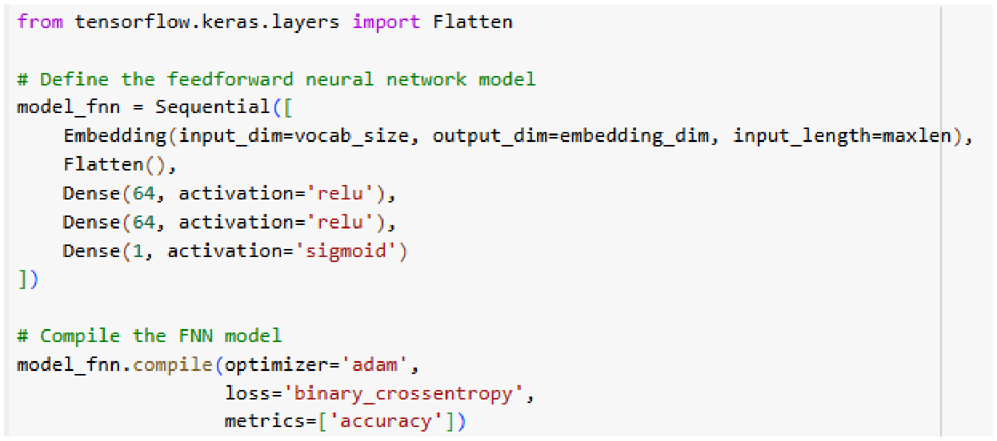
```
# Evaluate the model on the test set
test_loss_different_lr, test_accuracy_different_lr = model_different_lr.evaluate(x_test, y_test)
print(f'Test Loss with Different Learning Rate: {test_loss_different_lr}')
print(f'Test Accuracy with Different Learning Rate: {test_accuracy_different_lr}')

# Plot training & validation accuracy and loss values
plt.figure(figsize=(12, 4))
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 72ms/step - accuracy: 0.8758 - loss: 0.2911
Test Loss with Different Learning Rate: 0.29164841771125793
Test Accuracy with Different Learning Rate: 0.8763999938964844
```



These graphs show model performance with a different learning rate over 5 epochs. The training accuracy steadily increases, while validation accuracy improves quickly initially but plateaus after epoch 2. The loss graph shows both training and validation loss decreasing, with validation loss stabilizing around epoch 2. This suggests the new learning rate allows for faster initial learning and better generalization, but the model may be approaching its capacity to improve further on the validation set after epoch 2.

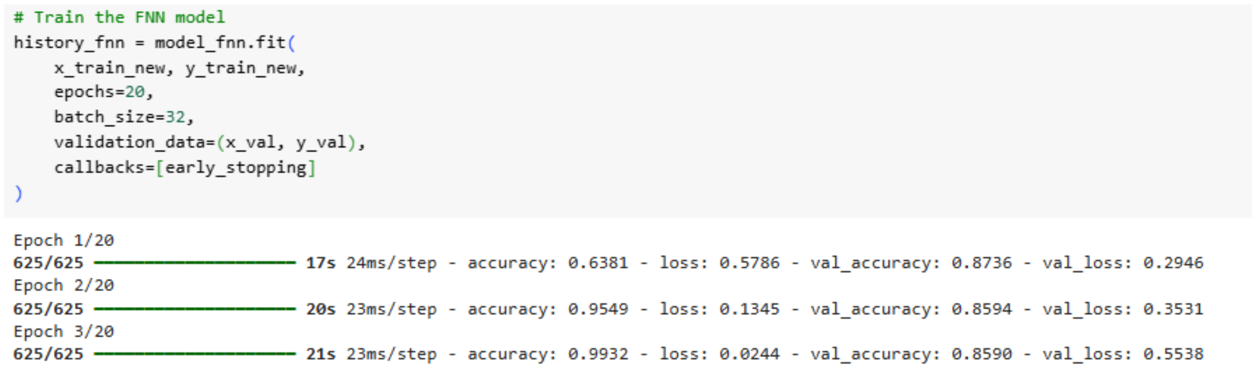
FeedForward Neural Network (FNN) Model



```python
from tensorflow.keras.layers import Flatten

# Define the feedforward neural network model
model_fnn = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the FNN model
model_fnn.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

The model_fnn is a feedforward neural network with an embedding layer to handle sequential input data, followed by a Flatten layer to reshape the data, and two fully connected Dense layers with ReLU activation. The final Dense layer uses a sigmoid activation for binary classification. This architecture is designed to learn complex representations and make predictions based on input sequences.

Tran the FeedForward Neural Network (FNN) Model

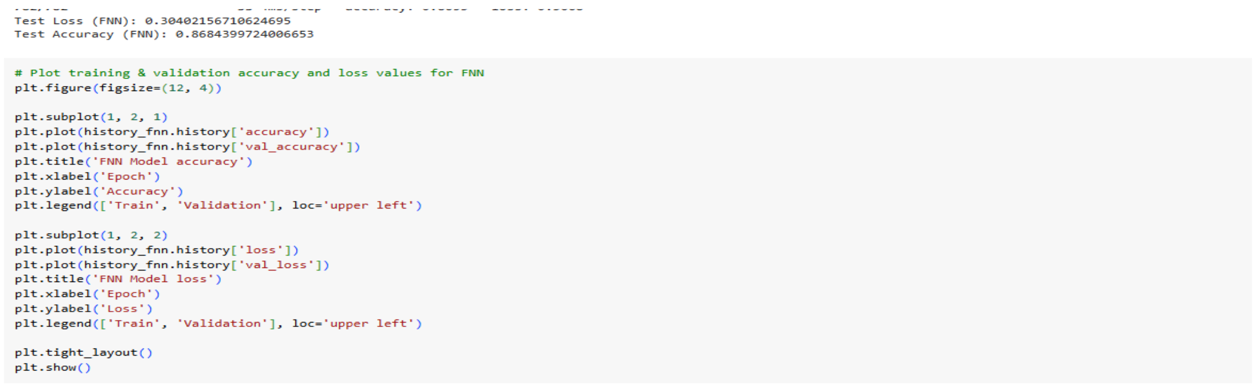


```python
# Train the FNN model
history_fnn = model_fnn.fit(
    x_train_new, y_train_new,
    epochs=20,
    batch_size=32,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping]
)
```

```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.6381 - loss: 0.5786 - val_accuracy: 0.8736 - val_loss: 0.2946
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 23ms/step - accuracy: 0.9549 - loss: 0.1345 - val_accuracy: 0.8594 - val_loss: 0.3531
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 21s 23ms/step - accuracy: 0.9932 - loss: 0.0244 - val_accuracy: 0.8590 - val_loss: 0.5538
```

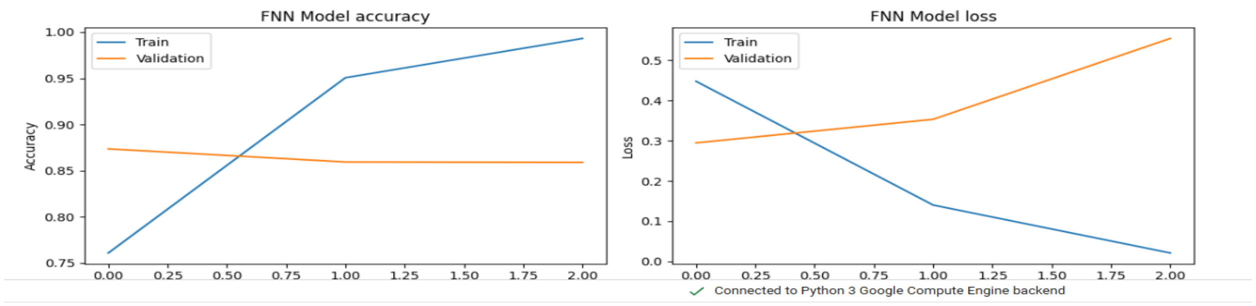
The output displays the first 3 epochs, showing improvements in training accuracy and decreases in training loss, while validation accuracy remains relatively stable. However, validation loss increases after the second epoch, potentially indicating the start of overfitting.

Evaluate the model



```
Test Loss (FNN): 0.30402156710624695
Test Accuracy (FNN): 0.8684399724006653
```

```
# Plot training & validation accuracy and loss values for FNN
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history_fnn.history['accuracy'])
plt.plot(history_fnn.history['val_accuracy'])
plt.title('FNN Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history_fnn.history['loss'])
plt.plot(history_fnn.history['val_loss'])
plt.title('FNN Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight_layout()
plt.show()
```



- **Accuracy:** The training accuracy steadily increases, while the validation accuracy seems to stabilize after a certain point. This suggests that the model is learning from the training data but might be overfitting (performing well on training data but not generalizing well to new data).
- **Loss:** The training loss decreases, indicating that the model is improving its predictions. The validation loss also decreases initially but then starts to increase, which again points to potential overfitting.

Overall, the model shows performance on the training data but might require further tuning or regularization to improve its generalization ability.

Compare the performance of the RNN with the FNN Model

To compare the performance of the RNN (Recurrent Neural Network) and the feedforward neural network (FNN) model

RNN Model Performance:

- Architecture:

Embedding → LSTM → Dropout → Dense → Dense

- Accuracy: RNNs generally achieved high training and validation accuracy, effectively learning from sequential data and capturing temporal dependencies.
- Training Time: RNNs require longer training times due to the complexity of processing sequences and learning long-term dependencies.
- Generalization: They usually provide good generalization on validation and test sets,

Feedforward Neural Network (FNN) Performance:

- Architecture:

Embedding → Flatten → Dense → Dense → Dense

- Accuracy: FNNs high accuracy
- Training Time: Generally faster to train compared to RNNs due to a simpler architecture and fewer parameters.

Conclusion

RNN:

Strengths: Better at handling sequential and time-series data, capturing dependencies over sequences.

Weaknesses: More complex and computationally expensive, risk of overfitting if not managed properly.

FNN:

Strengths: Simpler, faster to train, and easier to implement for non-sequential tasks.

Weaknesses: Limited in capturing sequential relationships and context, which can lead to reduced performance on tasks requiring understanding of order.

In general, RNNs performed better. However, for when computational resources are limited, FNNs can be a viable alternative, though they may not achieve the same performance on sequential data.