DEEP LEARNING FINAL PROJECT REPORT

GROUP 11

DECODING MOVIE OPINIONS WITH RNNs: ANALYSING SENTIMENT OF IMDB REVIEWS

AND

BATTLESHIP BOARD GAME: OPTIMAL STRATEGY FOR A COMPUTER (AI) TO PLAY

TEAM MEMBERS

SUBRAMANIAN ARUMUGAM

RIDA FATHIMA

AGASH SEKAR

ABINESH GANESAN

HARI PRIYA AVARAMPALAYAM MANOHARAN

PART -A

DECODING MOVIE OPINIONS WITH RNNs: ANALYSING SENTIMENT OF IMDB REVIEWS

Problem Statement:

The volume of data generated on the internet has increased exponentially, and with this, the need to process and analyse it has become paramount. One area where this is particularly important is in analysing user opinions and sentiments toward movies. Manually processing and interpreting a large volume of movie reviews to determine their overall sentiment is not only time-consuming but also prone to errors. Therefore, there is a need to develop an effective sentiment analysis model that can automatically analyse and classify movie reviews based on their overall sentiment.

Proposed Solution:

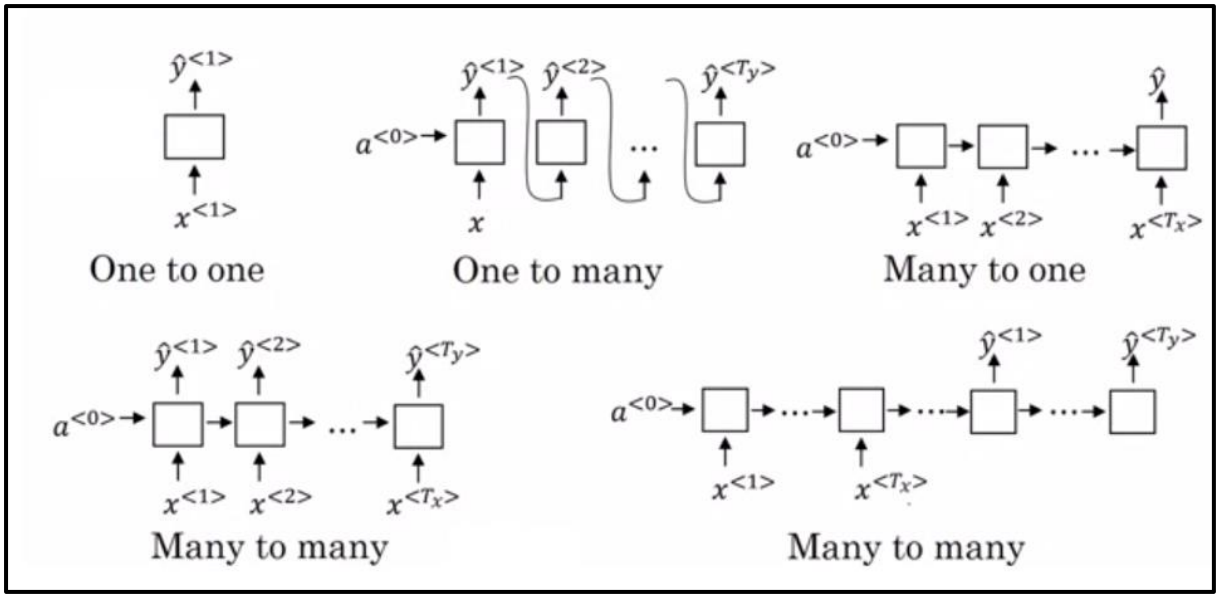
The proposed solution to the problem is to develop a sentiment analysis model that utilizes Natural Language Processing (NLP) and Recurrent Neural Networks (RNNs) to automatically classify IMDB movie reviews as positive or negative. The model will utilize NLP to pre-process the reviews, including steps such as tokenization, stop-word removal, and stemming. The pre-processed data will be then fed into an RNN-based model, which will learn to classify the reviews based on their overall sentiment. The study aims to evaluate the accuracy and performance of the proposed model by comparing it with existing sentiment analysis techniques. The goal is to provide insights into the practical application of the proposed sentiment analysis model for analyzing movie reviews.

Natural Language Processing (NLP):

Natural Language Processing (NLP) is a branch of artificial intelligence that focuses on the interaction between human language and computers. NLP techniques are used to analyze, interpret, and generate human language data, such as text and speech. With the vast amount of textual data generated on the internet, NLP has become increasingly important for various applications, including sentiment analysis, machine translation, chatbots, and speech recognition. In this study, NLP techniques will be utilized to pre-process the IMDB movie reviews, including tokenization, stop-word removal, and Lemmatization, to prepare the data for sentiment analysis using Recurrent Neural Networks (RNNs). The use of NLP in this study highlights its practical application in machine learning models that require natural language understanding.

RNN:

Recurrent Neural Networks (RNNs) are a type of neural network commonly used in Natural Language Processing (NLP) tasks, such as sentiment analysis, speech recognition, and language translation. RNNs are unique in that they can consider the sequence of data, such as words in a sentence or characters in a word, to learn patterns and relationships in the data. This makes them particularly useful for tasks where the input data has a temporal component, such as language data. In this study, an RNN-based model will be utilized for sentiment analysis of IMDB movie reviews. The model will be trained on the pre-processed data, including tokenized and stemmed words, to learn the patterns in the data and classify the reviews based on their overall sentiment. The use of RNNs in this study highlights their effectiveness in NLP tasks that require the analysis of sequential data.

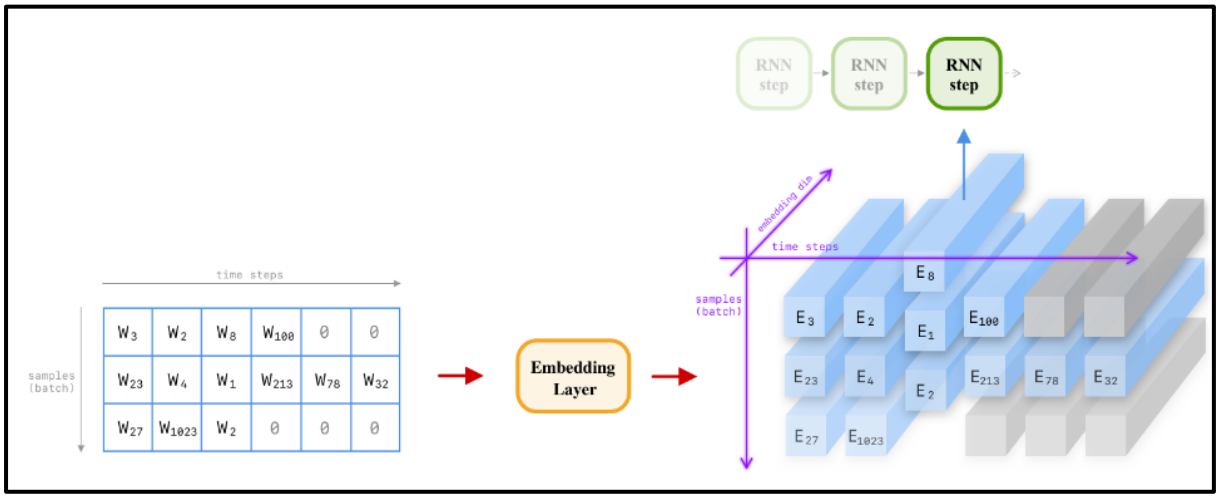


In deep learning, Recurrent Neural Networks (RNNs) are commonly used for Natural Language Processing (NLP) tasks, such as text classification, story generation, machine translation, and named entity recognition. Different types of RNN architectures can be utilized for different NLP tasks.

- The one-to-one RNN architecture is similar to a typical neural network but with added complexity. It is commonly used for simple classification tasks.
- The one-to-many RNN architecture is used for generating sequences of data, such as a story or music generation, by providing a single starting point.
- The many-to-one RNN architecture, which is commonly used for sentiment analysis, classifies input data based on the last (final) hidden state of the RNN.
- The many-to-many RNN architecture is utilized for named entity recognition, where each word belongs to a probability distribution of possible classes. This architecture is also used for machine translation and generating captions. The Seq2Seq architecture can be used to pass all words one by one, outputting the last cell state, which is then used to initialize a new RNN to generate the next word in the sequence. This process repeats until the desired output is generated or a stopping point is reached.

Training Data Example:

The below diagram shows an example of training data.



- Each BLUE column is a word representing a vector.
- The padding vector is a column of all zeros that are added to represent text data that has fewer words than expected. It is denoted by grey columns in the matrix representation, and the padding is defined by the padding parameter. Padding can be added either at the beginning or the end of the text data, as specified by the padding parameter. The padding vector ensures that all text data is represented as a fixed-length sequence that can be processed by machine learning algorithms.
- The X-axis represents the number of words in text data, such as a line, sentence, or document.
- The Y-axis represents the number of text data points to be processed, which determines the batch size.

LSTM:

Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) architecture that is commonly used in Natural Language Processing (NLP) tasks such as sentiment analysis. LSTM networks are designed to overcome the vanishing gradient problem in traditional RNNs by incorporating a memory cell and gates that allow the network to selectively forget or remember information over time. This memory mechanism makes LSTMs particularly useful for processing sequential data, such as text and has been shown to achieve state-of-the-art performance in sentiment analysis tasks using NLP. When applied to IMDB ratings, LSTMs can learn to analyze the sentiment of movie reviews and predict the overall rating based on the text input.

Dataset Description:

Dataset source: http://ai.stanford.edu/~amaas/data/sentiment

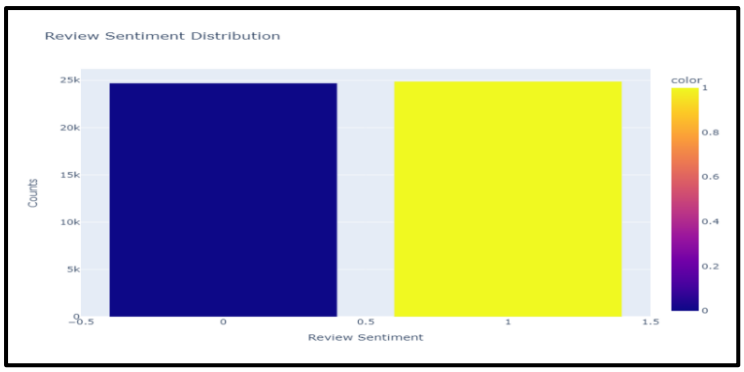
The IMDB dataset consists of 50,000 movie reviews that can be utilized for natural language processing and text analytics. It is a benchmark dataset for binary sentiment classification, with 25,000 reviews allocated for training and 25,000 for testing. The objective is to classify the reviews as either positive or negative using classification or deep learning algorithms.

Data Cleaning:

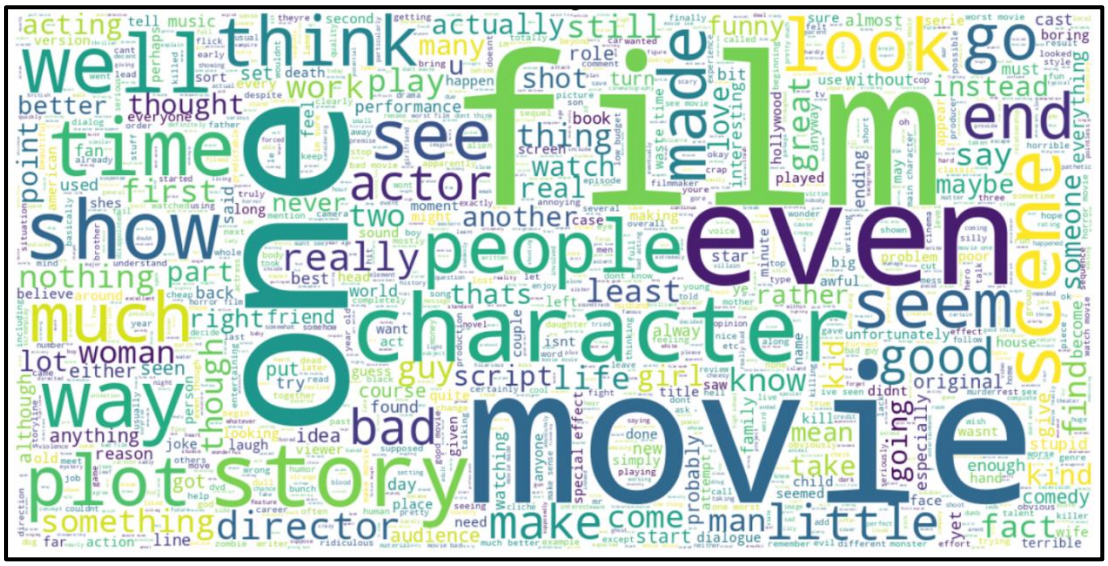
- **Removing null values:** This step involves identifying any missing or null values in your dataset and removing them. This is important because null values can cause errors in data analysis or machine learning models.
- **Removing duplicate values**: This step involves identifying and removing any duplicate rows or observations in your dataset. Duplicate values can skew your analysis or model results, and can also waste computational resources.
- **Removing stop words:** Stop words are common words that often appear in text data, such as "the," "and," and "is." Removing these words can improve the efficiency of natural language processing tasks and reduce noise in your data.
- **Lemmatization:** It is a technique in natural language processing that involves reducing words to their base or root form. This process eliminates redundancies and can provide a more accurate count of a word's frequency. Additionally, lemmatization can enhance the efficiency of NLP tasks. By reducing words to their base form, the number of unique words that need to be processed can be reduced, resulting in faster execution of certain NLP tasks. Overall, lemmatization plays a crucial role in optimizing various NLP processes, leading to improved accuracy and speed.

EDA:

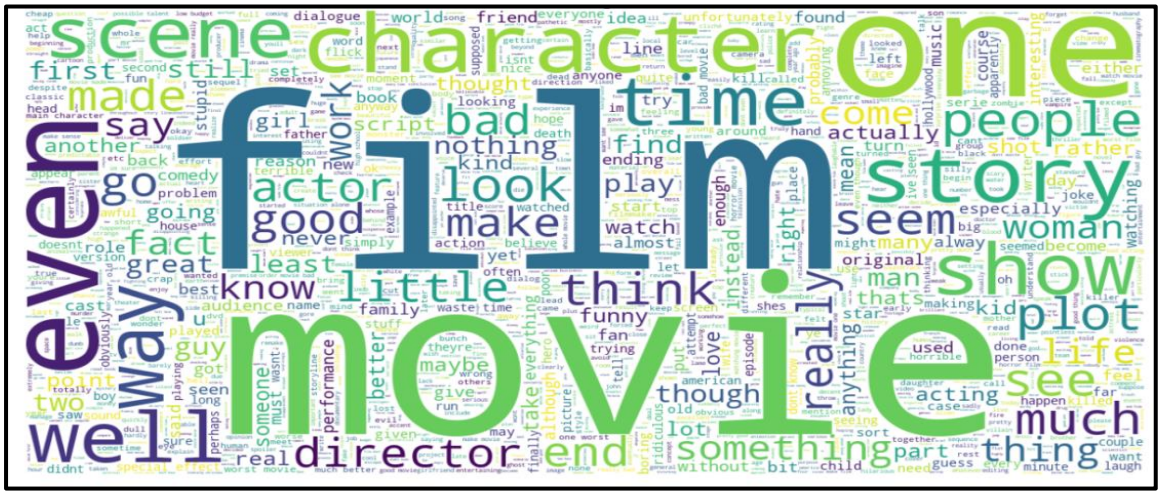
- Bar Chart to Visualize the Proportion of Positive and Negative Reviews



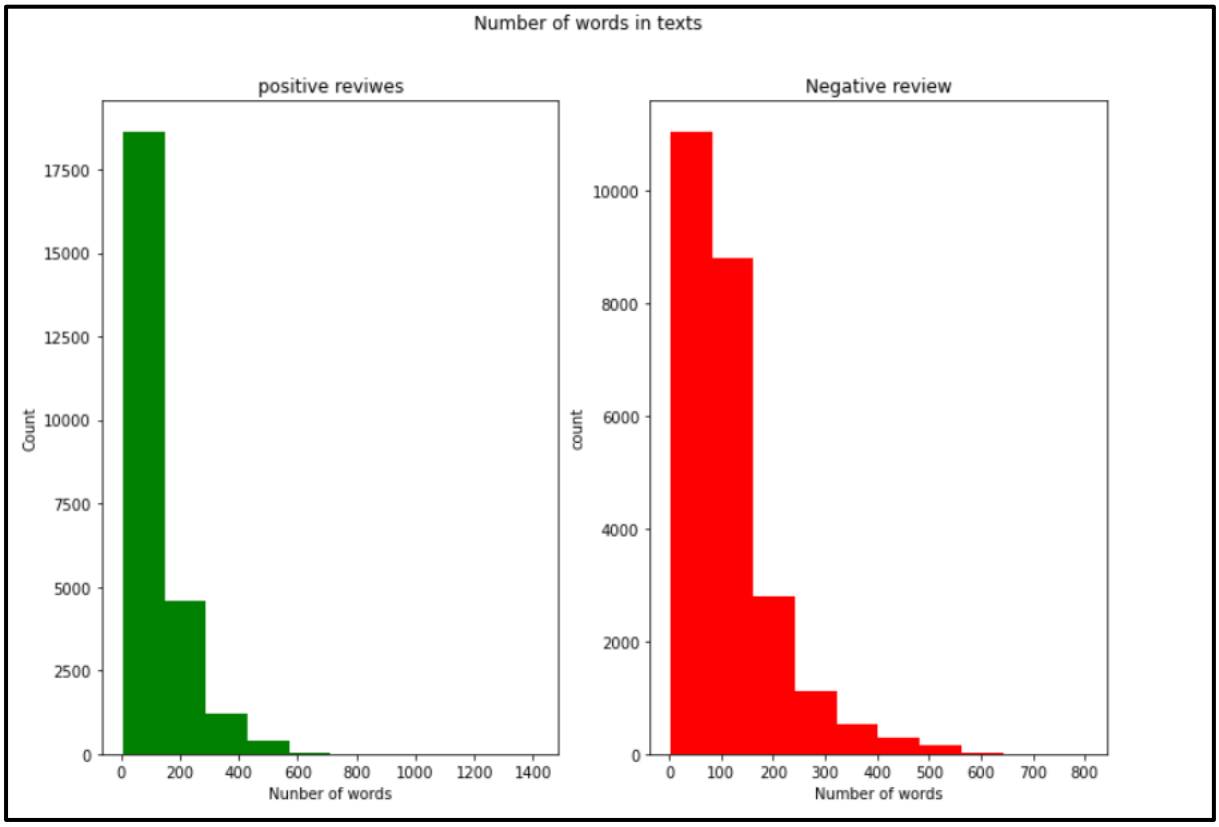
- Word Cloud to Visualize Highly Frequent words present in Negative Reviews



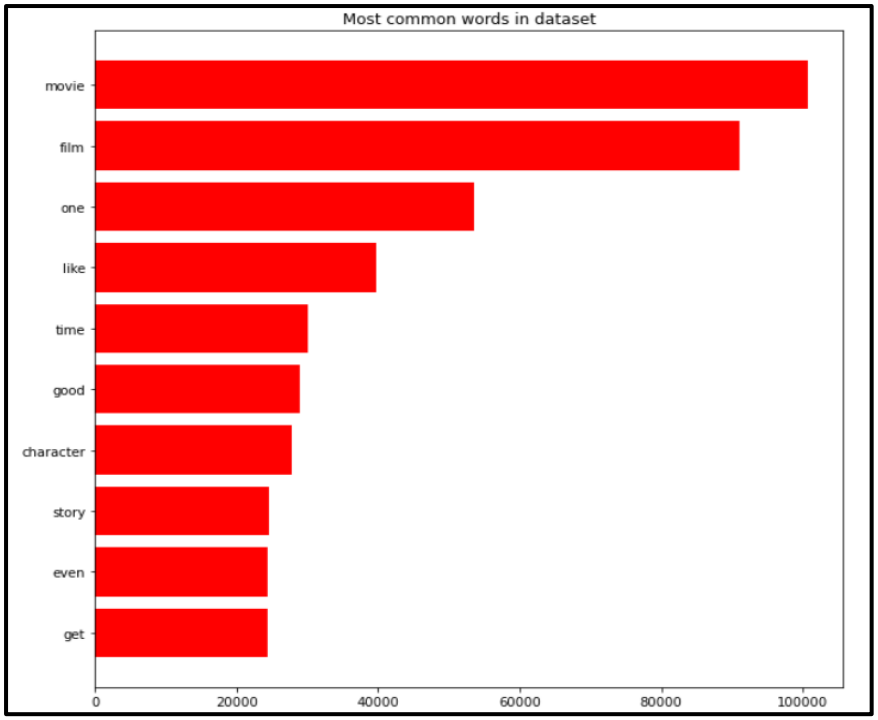
- Word Cloud to Visualize Highly Frequent words present in Positive Reviews



- The below graph shows the distribution of the number of words in positive and negative reviews.



- The below graph shows the distribution of the most common words in the dataset.



	words	count_
0	movie	100605
1	film	91130
2	one	53525
3	like	39746
4	time	30091
5	good	28905
6	character	27801
7	story	24600
8	even	24440
9	get	24328

Text Pre-processing:

The data cleaning steps were crucial in removing noise and irrelevant information from the text data, enabling the RNN model to learn more effectively and accurately predict the sentiment of IMDB reviews.

- **Changing text to lowercase:**

  In this step, we converted all the text to lowercase. This is done to ensure that the network treats the same word, regardless of its case, as the same word. For example, "movie" and "Movie" should be treated the same way by the network.

- **Removing HTML tags:**

  IMDb reviews can contain HTML tags, such as
 and <p>, which are not relevant for sentiment analysis. In this step, we removed these tags using regular expressions to clean up the text.

- **Removing URLs:**

  IMDb reviews can also contain URLs, which are not relevant for sentiment analysis. In this step, we removed any URLs using regular expressions.

  Overall, these pre-processing steps are essential to clean up the text data and remove any irrelevant information that may affect the accuracy of the sentiment analysis. By doing so, we can improve the performance of the RNN model in predicting the sentiment of IMDb reviews.

Custom Callback Function:

In this case, the on_epoch_end method is overridden to check if the accuracy metric has surpassed a threshold of 95%. If the condition is met, the training process is stopped by setting the stop training attribute of the model to True. This custom behaviour is useful when one wants to stop the training process early if the desired level of accuracy is achieved, which can save time and computational resources.

Pre-defined Call-backs:

A scheduler function is defined that takes two arguments, the current epoch number and the current learning rate, and returns the updated learning rate. The function returns a fixed learning rate of 0.01 for the first two epochs and then reduces the learning rate by a factor of 0.99 for each subsequent epoch. This strategy allows for a slower learning rate over time, which can help prevent overfitting and improve the generalization of the model.

Hyper Parameter Tuning:

Grid Search is the Technique we have used to fetch the best parameters for our Models. We first defined a grid of hyperparameters for our models to search over, and then trained the model with each combination of hyperparameters to evaluate its performance using a cross-validation technique.

Finally, we select the combination of hyperparameters that gives the best performance on the validation set. The best combination of hyperparameters is printed at the end of the Grid Search process. This combination can then be used to train a final model on the entire training set and evaluate its performance on a separate test set. We performed Grid Search on all three models we built.

Modelling:

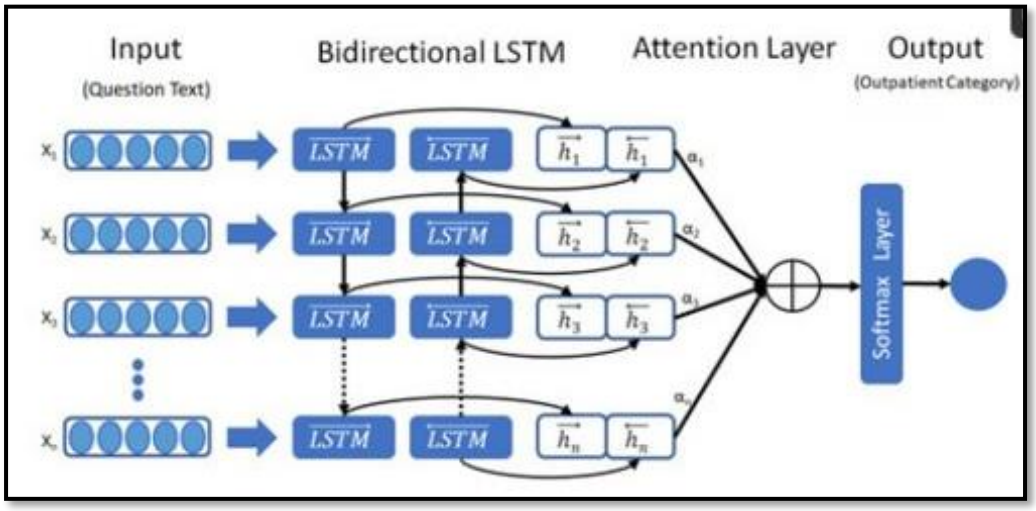
- **Model 1 – Bi-directional LSTM:**

The model used in the report is a sequential neural network built using the Kera's library with the TensorFlow backend. It consists of an embedding layer, a bidirectional LSTM layer, two dropout layers, and two dense layers. The embedding layer maps the input tokens to dense vectors, while the bidirectional LSTM layer processes the input sequences in both forward and backward directions. The dropout layers help prevent overfitting by randomly dropping out some neurons during training, and the dense layers use the rectified linear unit (ReLU) and sigmoid activation functions for non-linearity and output a binary sentiment prediction. The model was compiled with the binary cross-entropy loss function, Adam optimizer, and accuracy metrics. The model architecture and summary can be seen above.

- **Tokenization:**

We have performed Tokenization to break down a text into individual units, or tokens, such as words or sub-words.

- **Architecture:**



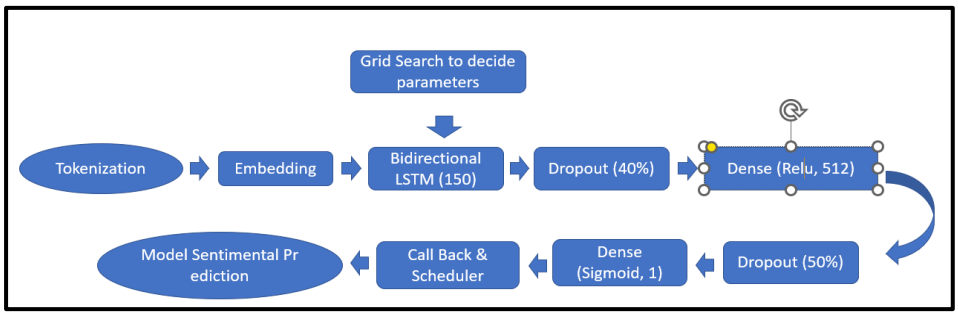
- **Grid Search:**

In our study, we investigated the effect of varying the number of neurons in the dense layer and LSTM units on the performance of our model. To identify the optimal number of neurons, we utilized the Grid search process. Specifically, we defined a dictionary called "param_grid," which outlined the specific values to be tested for the two hyperparameters, neurons and lstm_units. The values we tested included two different values for neurons (256 and 512) and two different values for lstm_units (100 and 150).

We evaluated the accuracy of all possible combinations of these values and determined the optimal set of parameters based on the accuracy results.

```
174/174 [==============================] - 32s 182ms/step - loss: 0.2399 - accuracy: 0.9006 - lr: 0.0099
Epoch 4/100
174/174 [==============================] - 33s 191ms/step - loss: 0.1508 - accuracy: 0.9408 - lr: 0.0098
Epoch 5/100
173/174 [============================>.] - ETA: 0s - loss: 0.0907 - accuracy: 0.9658Accuracy over 95%... Stopping training
174/174 [==============================] - 31s 180ms/step - loss: 0.0907 - accuracy: 0.9658 - lr: 0.0097
Epoch 1/100
174/174 [==============================] - 38s 197ms/step - loss: 0.5232 - accuracy: 0.7405 - lr: 0.0100
Epoch 2/100
174/174 [==============================] - 33s 188ms/step - loss: 0.3566 - accuracy: 0.8500 - lr: 0.0100
Epoch 3/100
174/174 [==============================] - 33s 188ms/step - loss: 0.2298 - accuracy: 0.9087 - lr: 0.0099
Epoch 4/100
173/174 [============================>.] - ETA: 0s - loss: 0.1327 - accuracy: 0.9515Accuracy over 95%... Stopping training
174/174 [==============================] - 35s 200ms/step - loss: 0.1326 - accuracy: 0.9515 - lr: 0.0098
Epoch 1/100
174/174 [==============================] - 36s 187ms/step - loss: 0.5242 - accuracy: 0.7375 - lr: 0.0100
Epoch 2/100
174/174 [==============================] - 32s 186ms/step - loss: 0.3609 - accuracy: 0.8454 - lr: 0.0100
Epoch 3/100
174/174 [==============================] - 32s 183ms/step - loss: 0.2258 - accuracy: 0.9111 - lr: 0.0099
Epoch 4/100
174/174 [==============================] - 31s 181ms/step - loss: 0.1411 - accuracy: 0.9464 - lr: 0.0098
Epoch 5/100
173/174 [============================>.] - ETA: 0s - loss: 0.0930 - accuracy: 0.9663Accuracy over 95%... Stopping training
174/174 [==============================] - 33s 192ms/step - loss: 0.0930 - accuracy: 0.9663 - lr: 0.0097
Epoch 1/100
260/260 [==============================] - 51s 182ms/step - loss: 0.5100 - accuracy: 0.7510 - lr: 0.0100
Epoch 2/100
260/260 [==============================] - 48s 184ms/step - loss: 0.3677 - accuracy: 0.8404 - lr: 0.0100
Epoch 3/100
260/260 [==============================] - 48s 183ms/step - loss: 0.2648 - accuracy: 0.8923 - lr: 0.0099
Epoch 4/100
260/260 [==============================] - 47s 181ms/step - loss: 0.1817 - accuracy: 0.9288 - lr: 0.0098
Epoch 5/100
260/260 [==============================] - ETA: 0s - loss: 0.1279 - accuracy: 0.9514Accuracy over 95%... Stopping training
260/260 [==============================] - 48s 186ms/step - loss: 0.1279 - accuracy: 0.9514 - lr: 0.0097
Best: 0.753153 using {'lstm_units': 200, 'neurons': 512}
```

➢ Summary:

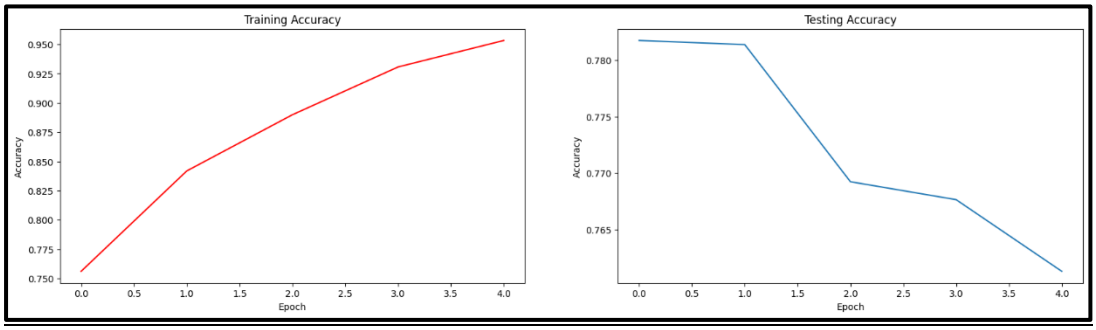



```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_16 (Embedding)    (None, 20, 64)            640000

 bidirectional_16 (Bidirecti  (None, 300)              258000
 onal)

 dropout_32 (Dropout)        (None, 300)               0

 dense_32 (Dense)            (None, 512)               154112

 dropout_33 (Dropout)        (None, 512)               0

 dense_33 (Dense)            (None, 1)                 513

=================================================================
Total params: 1,052,625
Trainable params: 1,052,625
Non-trainable params: 0
_________________________________________________________________
```

➢ Fitting Model-1:

```
Epoch 1/10
260/260 [==============================] - 55s 170ms/step - loss: 0.5033 - accuracy: 0.7562 - val_loss: 0.4624 - val_accuracy: 0.7818 - lr: 0.0100
Epoch 2/10
260/260 [==============================] - 47s 181ms/step - loss: 0.3677 - accuracy: 0.8419 - val_loss: 0.4627 - val_accuracy: 0.7814 - lr: 0.0100
Epoch 3/10
260/260 [==============================] - 42s 161ms/step - loss: 0.2667 - accuracy: 0.8899 - val_loss: 0.6123 - val_accuracy: 0.7692 - lr: 0.0099
Epoch 4/10
260/260 [==============================] - 45s 175ms/step - loss: 0.1795 - accuracy: 0.9307 - val_loss: 0.7512 - val_accuracy: 0.7676 - lr: 0.0098
Epoch 5/10
260/260 [==============================] - ETA: 0s - loss: 0.1247 - accuracy: 0.9533Accuracy over 95%... Stopping training
260/260 [==============================] - 43s 164ms/step - loss: 0.1247 - accuracy: 0.9533 - val_loss: 0.8799 - val_accuracy: 0.7613 - lr: 0.0097
```

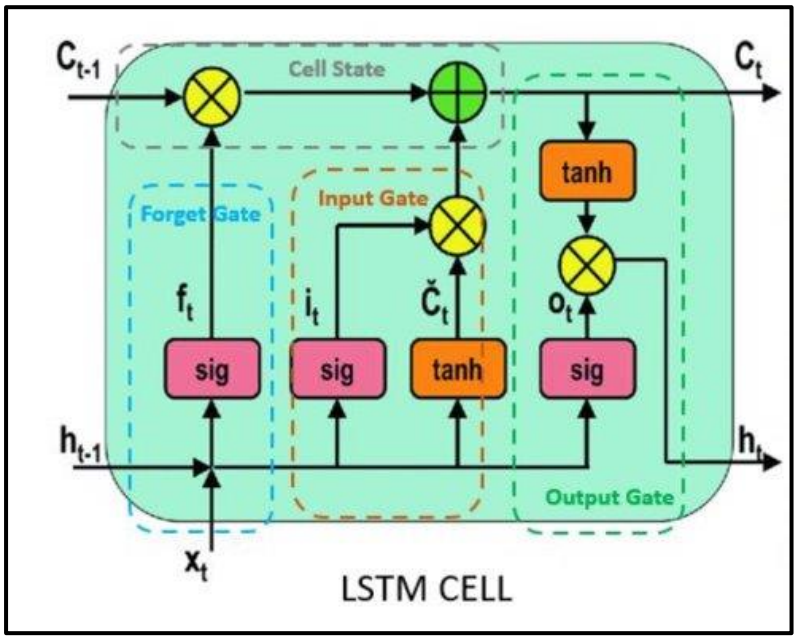
➢ **Plotting Training and Testing Accuracy v/s Epoch:**



Model 2 – LSTM (Unidirectional):

This Model uses an LSTM model with a different loss function and the activation function to classify movie reviews as positive or negative. The data is pre-processed using tokenization and padding. The model is then trained using Grid Search with different hyperparameters for neurons and epochs. The best hyperparameters are selected based on the highest accuracy score on the validation set. This model achieved an accuracy of 87% on the test set, which is a good improvement over the previous model. This shows the effectiveness of hyperparameter tuning in improving the performance of deep learning models.

➢ **Architecture:**



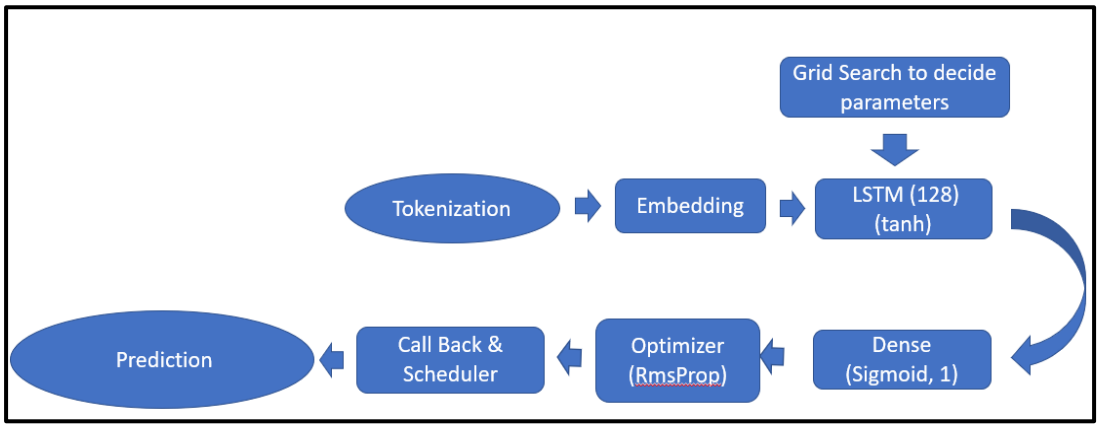
➢ **Grid Search:**

In our study, we investigated the impact of the number of neurons in the dense layer and epoch values on the performance of our model. To conduct this evaluation, we defined a dictionary named "param_grid" that specified the values to be tested for two hyperparameters: neurons and lstm_units. We set two values for neurons (64 and 128) and one value for epochs (3).

We assessed the accuracy of both combinations and identified the optimal parameters based on the accuracy results.

```
Epoch 3/3
207/207 [==============================] - 68s 329ms/step - loss: 0.1992 - accuracy: 0.9255 - val_loss: 0.3155 - val_accuracy: 0.8741 - lr: 0.0099
Epoch 1/3
207/207 [==============================] - 66s 310ms/step - loss: 0.5853 - accuracy: 0.6976 - val_loss: 0.3609 - val_accuracy: 0.8382 - lr: 0.0100
Epoch 2/3
207/207 [==============================] - 64s 310ms/step - loss: 0.2931 - accuracy: 0.8791 - val_loss: 0.3908 - val_accuracy: 0.8126 - lr: 0.0100
Epoch 3/3
207/207 [==============================] - 67s 323ms/step - loss: 0.1868 - accuracy: 0.9300 - val_loss: 0.3593 - val_accuracy: 0.8577 - lr: 0.0099
Epoch 1/3
310/310 [==============================] - 93s 295ms/step - loss: 0.5041 - accuracy: 0.7605 - val_loss: 0.3839 - val_accuracy: 0.8421 - lr: 0.0100
Epoch 2/3
310/310 [==============================] - 90s 289ms/step - loss: 0.2836 - accuracy: 0.8843 - val_loss: 0.2950 - val_accuracy: 0.8700 - lr: 0.0100
Epoch 3/3
310/310 [==============================] - 93s 300ms/step - loss: 0.1921 - accuracy: 0.9269 - val_loss: 0.3047 - val_accuracy: 0.8782 - lr: 0.0099
Best: 0.868869 using {'epochs': 3, 'neurons': 128}
```

- **Summary:**

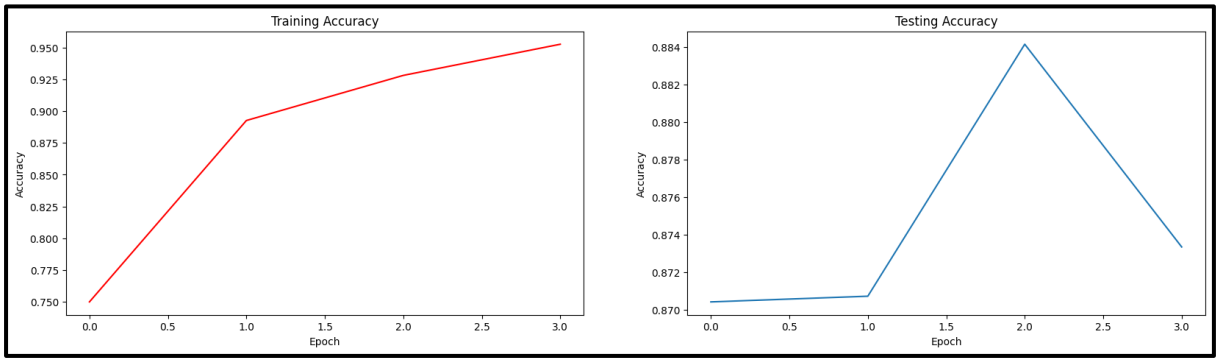



```
Model: "sequential_36"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_36 (Embedding)    (None, 100, 128)          2560000

 lstm_36 (LSTM)              (None, 128)               131584

 dense_36 (Dense)            (None, 1)                 129

=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
```

- **Fitting Model-2:**

```
Epoch 1/6
310/310 [==============================] - 103s 319ms/step - loss: 0.5097 - accuracy: 0.7498 - val_loss: 0.3087 - val_accuracy: 0.8704 - lr: 0.0100
Epoch 2/6
310/310 [==============================] - 97s 313ms/step - loss: 0.2703 - accuracy: 0.8927 - val_loss: 0.2965 - val_accuracy: 0.8707 - lr: 0.0100
Epoch 3/6
310/310 [==============================] - 97s 314ms/step - loss: 0.1893 - accuracy: 0.9282 - val_loss: 0.2819 - val_accuracy: 0.8841 - lr: 0.0099
Epoch 4/6
310/310 [==============================] - ETA: 0s - loss: 0.1356 - accuracy: 0.9528Accuracy over 95%... Stopping training
310/310 [==============================] - 98s 316ms/step - loss: 0.1356 - accuracy: 0.9528 - val_loss: 0.3391 - val_accuracy: 0.8733 - lr: 0.0098
```

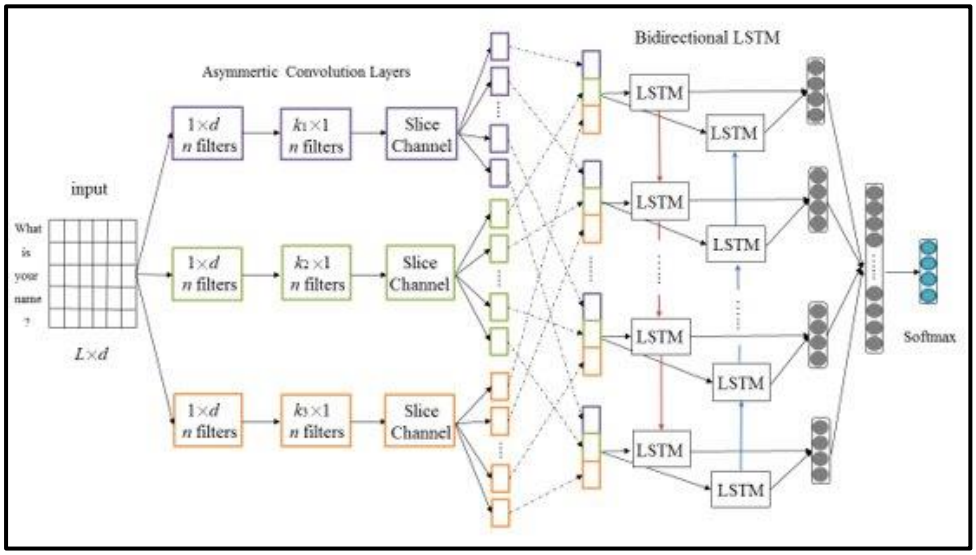
➢ **Plotting Training and Testing Accuracy v/s Epochs:**



Model 3 – LSTM with CNN:

This Model uses a CNN with a bidirectional LSTM model to classify movie reviews as positive or negative. The data is pre-processed using text vectorization and padding. The model is then trained using Grid Search with different hyperparameters for neurons and epochs. The best hyperparameters are selected based on the highest accuracy score on the validation set. This model achieved an accuracy of 88% on the test set, which is a significant improvement over the previous models. This shows the effectiveness of using a combination of convolutional and recurrent neural networks for natural language processing tasks.

➢ **Architecture:**



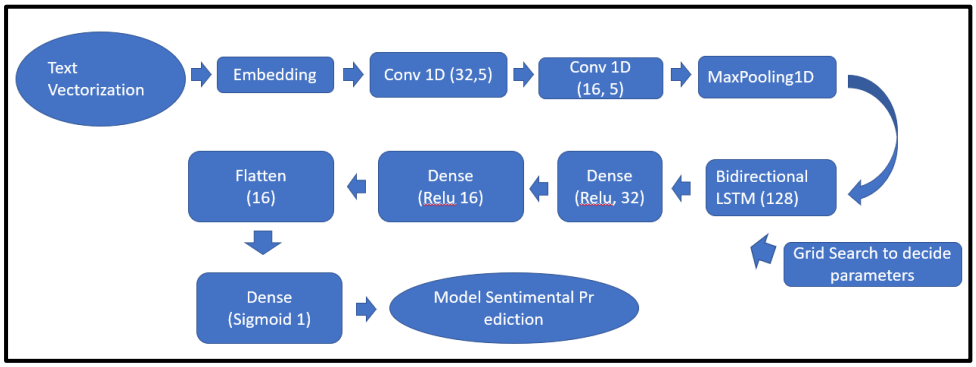
➢ **Grid Search:**

Our research aimed to analyze the effect of the number of neurons in the dense layer and epoch values on our model's performance. To facilitate this analysis, we created a dictionary named "param_grid" that outlined the values to be tested for two hyperparameters: neurons and lstm_units. Specifically, we tested two values for neurons (64 and 128) and one value for epochs (6).

We evaluated the accuracy of both combinations and determined the optimal set of parameters based on the accuracy results.

```
Epoch 4/6
174/174 [==============================] - 889s 5s/step - loss: 0.4272 - accuracy: 0.8530 - val_loss: 0.5494 - val_accuracy: 0.8230 - lr: 0.0098
Epoch 5/6
174/174 [==============================] - 886s 5s/step - loss: 0.3957 - accuracy: 0.8745 - val_loss: 0.5133 - val_accuracy: 0.8197 - lr: 0.0097
Epoch 6/6
174/174 [==============================] - 923s 5s/step - loss: 0.3666 - accuracy: 0.8808 - val_loss: 0.5064 - val_accuracy: 0.8247 - lr: 0.0096
Epoch 1/6
260/260 [==============================] - 1252s 5s/step - loss: 0.5233 - accuracy: 0.7428 - val_loss: 0.3967 - val_accuracy: 0.8396 - lr: 0.0100
Epoch 2/6
260/260 [==============================] - 1243s 5s/step - loss: 0.3540 - accuracy: 0.8684 - val_loss: 0.6247 - val_accuracy: 0.7423 - lr: 0.0100
Epoch 3/6
260/260 [==============================] - 1307s 5s/step - loss: 0.3589 - accuracy: 0.8681 - val_loss: 0.3950 - val_accuracy: 0.8501 - lr: 0.0099
Epoch 4/6
260/260 [==============================] - 1240s 5s/step - loss: 0.2604 - accuracy: 0.9147 - val_loss: 0.3838 - val_accuracy: 0.8564 - lr: 0.0098
Epoch 5/6
260/260 [==============================] - 1238s 5s/step - loss: 0.3470 - accuracy: 0.8773 - val_loss: 0.5246 - val_accuracy: 0.8065 - lr: 0.0097
Epoch 6/6
260/260 [==============================] - 1227s 5s/step - loss: 0.3303 - accuracy: 0.9013 - val_loss: 0.4554 - val_accuracy: 0.8592 - lr: 0.0096
Best: 0.701549 using {'epochs': 6, 'neurons': 128}
```

➢ Summary:

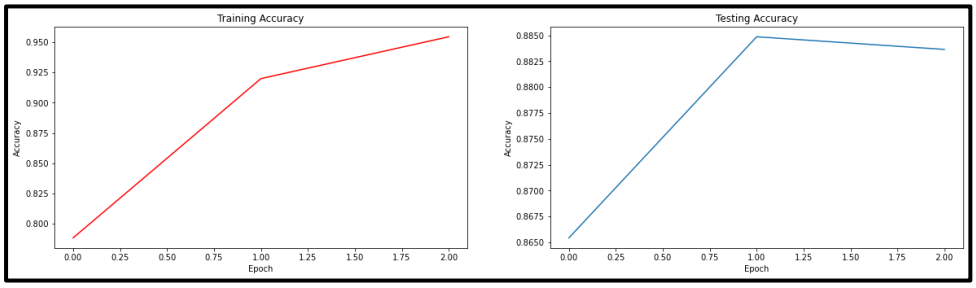



```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 text_vectorization_7 (TextV  (None, 512)              0
 ectorization)

 embedding_5 (Embedding)     (None, 512, 256)          29163520

 conv1d_10 (Conv1D)          (None, 512, 32)           40992

 conv1d_11 (Conv1D)          (None, 512, 16)           2576

 max_pooling1d_5 (MaxPooling  (None, 512, 16)          0
 1D)

 bidirectional_5 (Bidirectio  (None, 128)              41472
 nal)

 dense_15 (Dense)            (None, 32)                4128

 dense_16 (Dense)            (None, 16)                528

 flatten_5 (Flatten)         (None, 16)                0

 dense_17 (Dense)            (None, 1)                 17

=================================================================
Total params: 29,253,233
Trainable params: 29,253,233
Non-trainable params: 0
```

➢ Fitting Model-2:

```
Epoch 1/6
1039/1039 [==============================] - 493s 471ms/step - loss: 0.4465 - accuracy: 0.7884 - val_loss: 0.3344 - val_accuracy: 0.8654
Epoch 2/6
1039/1039 [==============================] - 477s 459ms/step - loss: 0.2221 - accuracy: 0.9200 - val_loss: 0.2969 - val_accuracy: 0.8849
Epoch 3/6
1039/1039 [==============================] - ETA: 0s - loss: 0.1426 - accuracy: 0.9545Accuracy over 95%... Stopping training
1039/1039 [==============================] - 465s 448ms/step - loss: 0.1426 - accuracy: 0.9545 - val_loss: 0.3439 - val_accuracy: 0.8836
```

➢ **Plotting Training and Testing Accuracy v/s Epoch:**



Model Comparison:

S.no	Models	Test Accuracy	Epochs
1.	Bi-directional LSTM	76.13 %	10
2.	LSTM (unidirectional)	87.33 %	6
3.	LSTM with CNN	88.36%	6

Conclusion:

Based on the provided information, we can make the following conclusions:

- **The Bi-directional LSTM model** has the lowest test accuracy **(76.13%)** among the three models.
- **LSTM unidirectional model** has a higher test accuracy **(87.33%)** than the Bi-directional LSTM model.
- **LSTM with the CNN model** has the highest test accuracy **(88.36%)** among the three models

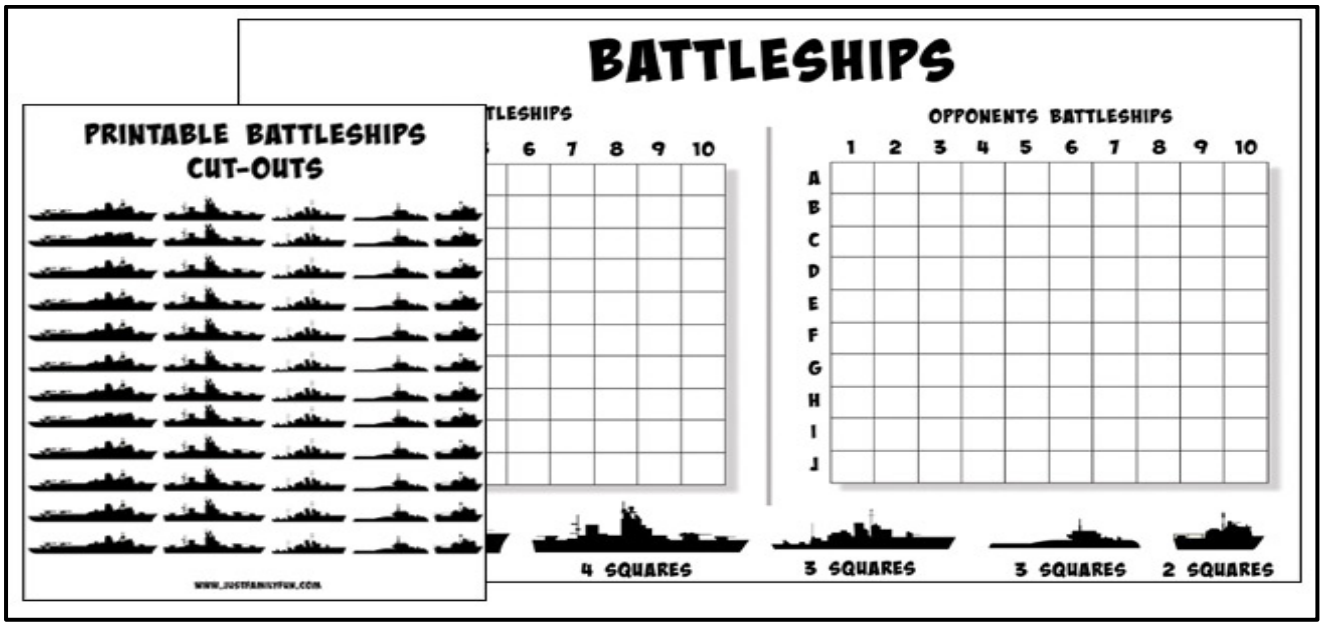
PART -B

REINFORCEMENT LEARNING

BATTLESHIP BOARD GAME: OPTIMAL STRATEGY FOR A COMPUTER (AI) TO PLAY

Battleship is a classic board game that has been popular for decades. It is a strategy-based game that requires players to think tactically and make strategic moves to win. The game is played on a grid, which is typically 10x10, and each player places their ships on their grid in a way that is hidden from their opponent. The goal of the game is to sink all the opponent's ships before they sink yours.

To play Battleship, each player takes turns guessing where the opponent's ships are located on their grid. If a player guesses correctly, they score a hit, and the opponent must mark that location on their grid as a hit. If a player misses, the opponent marks the location as a miss. The game continues until one player sinks all the opponent's ships or until both players agree to end the game. Battleship is a game of strategy and skill, and it can be played by players of all ages and skill levels.



DETAILED DESCRIPTION:

Actions:

Place Ships: Each player places a set of ships of varying sizes on their grid.

Guess: Each player takes turns guessing the location of the other player's ships on their grid.

Hit: If a player correctly guesses the location of an opposing ship, it is considered a hit.

Miss: If a player incorrectly guesses the location of an opposing ship, it is considered a miss.

Sunk: If all the spaces occupied by a ship are hit, the ship is sunk.

States:

Ship Placement: The placement of each player's ships on their grid.

Guesses: The previous guesses made by each player.

Hits: The previous hits made by each player.

Misses: The previous misses made by each player.

Sunks: The previous sinks made by each player.

Rewards/Penalties:

Reward: The reward for the AI player is for successfully hitting and sinking the opposing player's ships.

Penalty: The penalty for the AI player is for making an incorrect guess, which results in a miss and wastes a turn.

Other Game Mechanics:

Randomness: The game involves an element of randomness as players do not know the location of the opposing player's ships.

Strategy: The game involves strategic thinking and decision-making, such as where to place ships and where to guess.

HYPOTHESIS:

Hypothesis for Action Rule:

The hypothesis for action rules in the game Battleship could be a probabilistic decision-making process. The AI player could use information from previous guesses, hits, and sinks to calculate the probability of the location of the opposing player's ships.

The AI player would then guess the location with the highest probability of containing a ship. This strategy would help the AI player to sink the opposing player's ships and increase the chance of winning the game.

The AI player may also use a pattern recognition algorithm to identify potential ship configurations and locations. A neural network could be used to implement this strategy, with activation functions such as softmax and sigmoid.

Hypothesis for Reward:

The hypothesis for the reward system in the Battleship game is that the AI player will receive a reward based on the number of ships sunk. The more ships the AI player sinks, the higher the reward. The reward could be in the form of points earned for each successful round, with a cumulative value calculated at the end of the game.

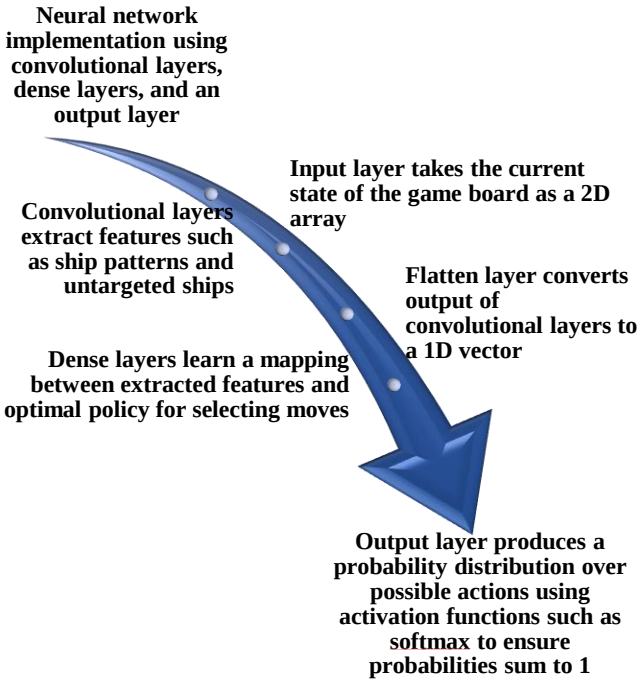
The ultimate goal for the AI player is to maximize their reward by sinking all of the opposing player's ships as quickly as possible. This reward system provides the necessary motivation for the AI player to focus on sinking ships and winning the game.

Hypothesis for State Distribution:

The AI player's state distribution in Battleship could be influenced by their previous guesses, hits, and sinks. For example, if the AI player has already sunk a ship, the probability of the opposing player placing a ship in that area would be lower.

The AI player's state distribution is dynamic and can change based on new information and events in the game.

Layers and Activation Functions:



For the neural network implementation of the AI player's decision-making process, we can use a combination of convolutional layers, dense layers, and an output layer.

*The **input layer** would take in the current state of the game board as a 2D array. Each element in the array represents a square on the board, and its value is either 0 if the square is empty, 1 if the square has a ship, or -1 if the square has been targeted but no ship was found.*

*The **convolutional layers** would be responsible for extracting features from the game board, such as identifying ship patterns and detecting untargeted ships. The output of the convolutional layers would be a set of feature maps.*

*The **flattened layer** would take the output of the convolutional layers and flatten it into a 1D vector, making it easier to pass the features on to the next layer.*

*The **dense layers** would learn a mapping between the features extracted from the game board and the optimal policy for selecting moves. They would approximate the Q-values of the possible actions for each state of the game. We can experiment with different architectures and activation functions, such as ReLU or Tanh, to find the one that works best for our implementation.*

*The **output layer** would take the Q-values as input and produce the final output of the neural network, which is a probability distribution over the possible actions (i.e., the locations on the board to shoot*

at). We can use activation functions such as softmax to normalize the output and ensure that the sum of the probabilities is 1.

FINAL REWARD:

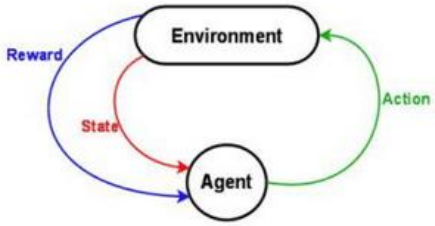
The final reward in the game of Battleship can be based on a binary output, such as winning or losing the game, or a cumulative output, such as the number of ships sunk in each round.

The optimal reward system can depend on various factors, such as the AI player's objectives, game setup, and the balance between exploration and exploitation.

The reward system should incentivize the AI player to achieve the desired outcome in the most efficient way possible, while also considering other factors such as the speed of sinking ships and the number of successful rounds.

ANALYSIS PROCEDURE:

- **Define the environment:** Define the battleship game board, including the number of rows and columns and the locations of the ships. Define the rules of the game, such as the number of ships, their sizes, and the number of shots allowed per turn.
- **Design the agent:** Use a neural network to train the agent to learn an optimal policy for selecting moves. Define the input and output layers of the neural network and select appropriate activation functions for each layer.



The learning agent learns by interacting with the environment and then figures out how to best map states to actions. The typical setup involves an environment, an agent, states, and rewards.
The agent is then tested against other opponents to evaluate its performance, and improvements are made to the reward function and hyperparameters as necessary.
Finally, the agent is deployed to play the game in a real-world environment, and its performance is continuously monitored and updated.
By following this procedure, we can create a powerful AI system that can play Battleship at a high level of proficiency.

- **Define the rewards:** Define a reward function that incentivizes the agent to win the game as quickly as possible. Use a positive reward for hitting a ship and a negative reward for missing a shot. Define the final reward system, which could be based on the number of rounds won or lost or a binary output of winning or losing.
- **Train the agent:** Use reinforcement learning algorithms such as Q-learning, SARSA, or policy gradients to train the agent on the battleship game. Use a simulation environment to train the agent without having to play the game manually. Tune hyperparameters such as the learning rate, discount factor, and exploration rate to optimize the agent's performance.

The most common technical approach is Q-learning. Here, our neural network acts as a function approximator for a function Q, where Q(state, action) returns a long-term value of the action given the current state. The simplest way to use an agent trained from Q-learning is to pick the action that has the maximum Q-value. The Q represents the "quality" of some move given a specific state

- **Test the agent:** Once the agent is trained, test it against a human player or a random agent to evaluate its performance. Use techniques such as Monte Carlo simulations to estimate the agent's win rate against other opponents. Evaluate the agent's performance metrics such as win rate, average reward, and convergence rate.
- **Improve the agent:** If the agent is not performing well, refine the reward function or adjust the hyperparameters of the neural network to improve its performance. Use techniques such as experience replay and target networks to stabilize the agent's learning and prevent overfitting.
- **Deploy the agent:** Deploy the agent to play battleship games in a real-world environment, such as an online gaming platform. Continuously monitor its performance to ensure it is working correctly and make necessary updates to the system. Consider expanding the agent's capabilities to handle more complex battleship scenarios, such as dynamic board configurations or multiple players.

Overall, this analysis procedure provides a framework for designing and implementing an effective reinforcement learning agent for the battleship game.