Comparative Analysis of RNN, LSTM, and GRU Models for Sentiment Analysis:

Performance Evaluation and Hyperparameter Optimization

African Institute for Mathematical Sciences CAMEROON

Course: Deep Deep Learning Methods and Algorithms

Under the supervision of

Paulin Melatagia Yonta, Ph.D.

Members:

1. Abibou Moustapha Motapon Ndam

2 Elizabeth Mwania

3 Mona Hussein Ali Edris

4 Oumar Moussa

5 Fred Okondza

Abstract

In this deep learning era, NLP is a thriving field that see lots of different advancements. One of them is sentiment analysis. In this study, we have applied an efficient Neural Language Model approach to compare three neural networks on Sentiment Analysis that relies only on unsupervised word representation inputs. The three models: -

a) **Model 1** that employs Embedding layer followed by a simple RNN layer followed by a fully connected layer.
b) **Model 2** that employs Embedding layer followed by an LSTM layer followed by a fully connected layer.
c) **Model 3** that employs Embedding layer followed by a GRU layer followed by a fully connected layer.

We also applied word embedding technique using Word2Vector, which has improved the model to capture syntactic and semantic word relationships. This project compares the three model on their performance in different aspects.

To select the best model, we have used Keras Tuner for hyperparameter optimization.

1. INTRODUCTION

Sentiment analysis, a key task in natural language processing (NLP), aims to determine the emotional tone of text, such as positive, negative, or neutral. It has widespread applications in areas like customer feedback analysis, social media monitoring, and market research. While traditional methods like rule-based systems and machine learning models have been used, deep learning approaches, particularly Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs), have shown superior performance due to their ability to capture contextual and sequential information. This report compares the performance of RNN, LSTM, and GRU models for sentiment analysis, evaluates the impact of hyperparameter tuning, and identifies the best-performing model for this task. The findings aim to provide insights into model selection and optimization for sentiment analysis applications.

2. LITERATURE REVIEW

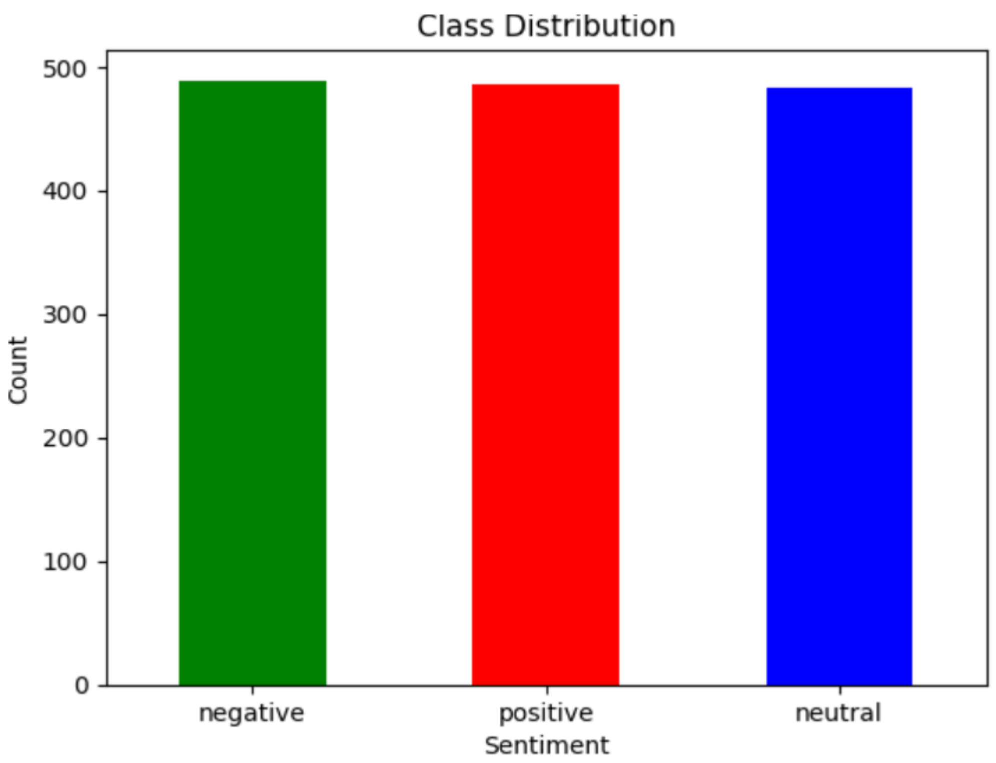
Sentiment analysis has evolved significantly from early rule-based and machine learning approaches to advanced deep learning models like RNNs, LSTMs, and GRUs. Traditional methods, such as Naive Bayes and SVM, relied on handcrafted features like bag-of-words but struggled with context and semantics. The introduction of word embeddings (e.g., Word2Vec) and recurrent architectures, particularly LSTMs and GRUs, revolutionized the field by capturing long-term dependencies and contextual nuances in text. Recent advancements, such as Transformer-based models (e.g., BERT), have further pushed the boundaries of sentiment analysis by leveraging large-scale pretraining and self-attention mechanisms. Hyperparameter tuning has also become a critical step in optimizing model performance, as demonstrated in this report, where LSTM outperformed RNN and GRU after tuning. This aligns with literature emphasizing LSTM's effectiveness in handling sequential data and its superiority in sentiment analysis tasks.

3. METHODOLOGY AND APPROACH

The methodology for semantic analysis is as follows: -

3.1. Getting Data

The dataset used in this study was generated using ollama LLM (1458 entries) with a variety of contexts to ensure diversity in sentiment representation. It contains labelled textual data categorized into three sentiment classes: positive, negative, and neutral. The dataset was curated to include sentences from domains such as business, healthcare, education, social media, and customer support, making it well-suited for general sentiment analysis tasks.



3.2. Data Preprocessing

To ensure high-quality input for model training, several preprocessing steps were applied:

1. **Data Cleaning**:
   - Removed punctuation, special characters, and unnecessary spaces.
   - Converted text to lowercase to maintain uniformity.
2. **One hot encoding:**
   We labelled our classes as follows
   [1,0,0] for positive, [0,1,0] for negative, [0,0,1] for neutral
3. **Tokenization**:
   - Split sentences into individual words (tokens) for further analysis.
4. **Stopword Removal**:
   - Eliminated common words (e.g., 'the', 'is', 'and') that do not contribute to sentiment.
5. **Lemmatization**:
   - Converted words to their base forms to reduce dimensionality (e.g., 'running' to 'run').
6. **Data Splitting**:
   - The dataset was divided into training and testing subsets using an 80-20 split to evaluate model performance.
7. **Word Embeddings**:
   - Used pre-trained Word2Vec embeddings to convert text into numerical vectors for input into neural network models.

3.3. Model Architecture

Recurrent Neural Networks (RNN) are to the rescue when the sequence of information is needed to be captured Unlike neural network where the information flows in one direction from input to output, in RNN information is fed back into the system after each hence it remembers past sequences along with current input which makes it capable to capture context rather than just individual words.

The model architecture looks like:

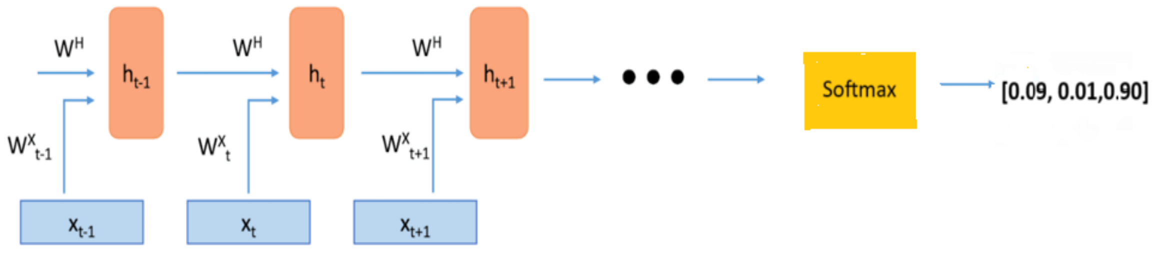


Figure 1 Many to One RNN Model Architecture

Where **h_t**: RNN layer at time t.
X_t: Word embeddings which acts as input for RNN layer at time t.
W: Represent weights.

The RNN model includes the following layers:

3.3.1. Input Layer

The input layer receives the tokenized text data, where each word is represented by an integer index. Input Shape (MAX_SEQUENCE_LENGTH,) is the input sequence of integers, where MAX_SEQUENCE_LENGTH is the maximum length of a sentence after padding (e.g., 50).

Example: If the sentence is "I love this product," it is tokenized and padded to a fixed length (e.g., [4, 12, 7, 23, 0, 0, ..., 0])

3.3.2. Embedding layer

The embedding layer will convert the integer-encoded words into dense vectors of fixed size (embeddings). It has three arguments. The input dimension in our case is 300 words. The output dimension aka the vector space in which words will be embedded. In our case we have chosen 300 dimensions so a vector of the length of 300 to hold our word coordinates. The Weights, in this case Pre-trained Word2Vec embeddings are used, and the layer is set to trainable=False to avoid updating the embeddings during training. We want these vectors to be created in such a way that they somehow represent the word and its context, meaning, and semantics. For example, we'd like the vectors for the words "love" and "adore" to reside in relatively the same area in the vector space since they both have similar definitions and are both used in similar contexts.

Context is also very important when considering grammatical structure in sentences. Most sentences will follow traditional paradigms of having verbs follow nouns, adjectives precede nouns, and so on. For this reason, the model is more likely to position nouns in the same general area as other nouns. The layer takes in a large dataset of sentences and outputs vectors for each unique word in the corpus.

3.3.3. RNN/GRU/LSTM Layer

This layer contains cells which processes the sequence of word embeddings to capture temporal dependencies in the text. RNNs are basic, while LSTMs and GRUs are more advanced variants designed to handle long-term dependencies.

Parameters:

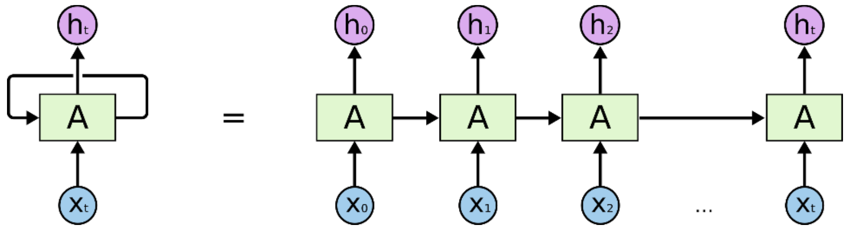
- **Units:** Number of hidden units in the layer (e.g., 256 before tuning, 128 for RNN, and 256 for LSTM/GRU after tuning).
- **Dropout:** Fraction of units to drop during training (e.g., 0.2 before tuning, 0.4 for LSTM/GRU after tuning).
- **Recurrent Dropout:** Fraction of recurrent connections to drop during training (e.g., 0.2).

Output Shape:
None, 256): The output is a single vector of size 256 (number of units) for each sequence

These Architectures are explained below:

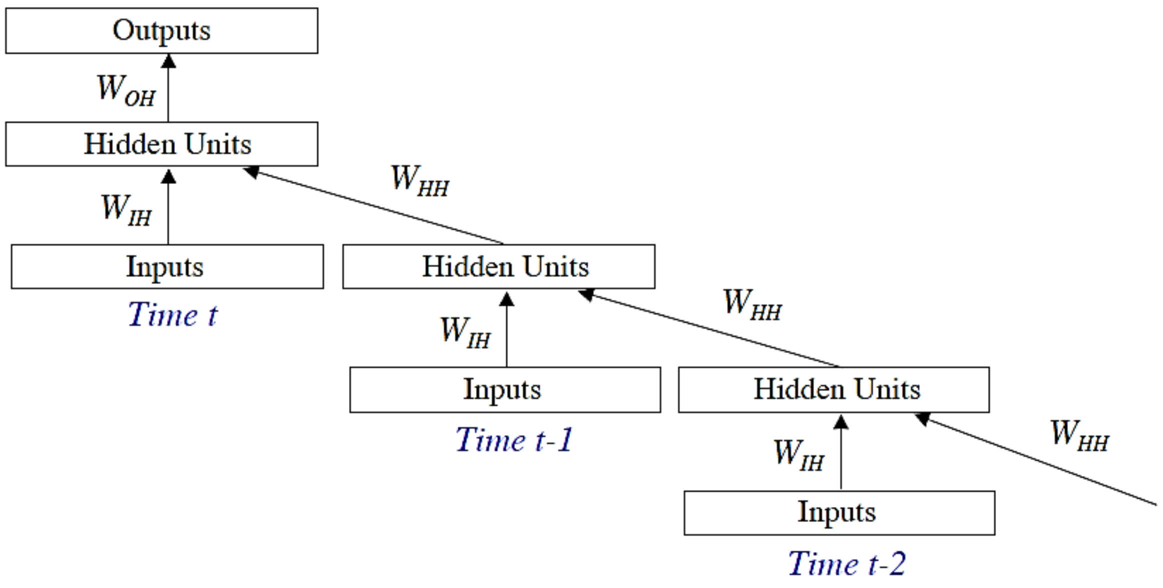
3.3.4. Simple RNN layer

The unique aspect of NLP data is that there is a temporal aspect to it. Each word in a sentence depends greatly on what came before and comes after it. In order to account for this dependency, we use a **many-to-one recurrent neural network**.



In the above diagram, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

The state space model: -



- Suppose inputs and outputs are the vectors $\boldsymbol{x}(\boldsymbol{t})$and $\boldsymbol{y}(\boldsymbol{t})$
- the three connection weight matrices are **W$_{IH}$**, **W$_{HH}$** and **W$_{HO}$**, and
- the hidden and output unit activation functions are **f$_H$** and **f$_O$**, then the behaviour of the recurrent network can by the pair of non-linear matrix equations:
  $h(t) = f_H(w_{IH}x(t) + w_{HH}h(t-1))$
  $y(t) = f_O(w_{HO}h(t))$
  Here, the state is defined by the set of hidden unit activations h(t).
  In my model the output is a binary variable with outputs 0 (Negative) and 1(Positive).

3.3.5. LSTM

LSTM is a type of recurrent neural network (RNN) designed to solve the vanishing gradient problem, making it more effective for learning long sequences than traditional RNNs and other sequence models. Its name reflects its ability to maintain short-term memory over long-time steps, inspired by concepts from cognitive psychology. LSTM overcomes the vanishing gradient problem by using memory gates that regulate the flow of information through the network, allowing gradients to persist over long time sequences without significant attenuation.

The LSTM cell regulates information flow through three gates:

- **Forget Gate**: Retains or discards past information:
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
- **Input Gate**: Incorporates new information:
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \quad \tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$
- **Cell State Update**: Combines retained memory and new input:
$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t$$
- **Output Gate & Hidden State**: Controls and transforms output:
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o), \quad h_t = o_t \otimes \tanh(C_t)$$

3.3.6. GRU

GRU use gating mechanisms to selectively update the hidden state of the network at each time step. The gating mechanisms are used to control the flow of information in and out of the network. The GRU has two gating mechanisms, called the reset gate and the update gate.

One State
- Hidden State $h_t = (1 - z_t) \otimes h_{t-1} \oplus z_t \otimes \tilde{h}_t$

Two Gates
- Update Gate $Z_t = \sigma(W_z \cdot x_t + U_z \cdot h_{t-1})$
- Reset Gate $r_t = \sigma(W_r \cdot x_t + U_r \cdot h_{t-1})$

3.3.7. Fully Connected layer with dropouts

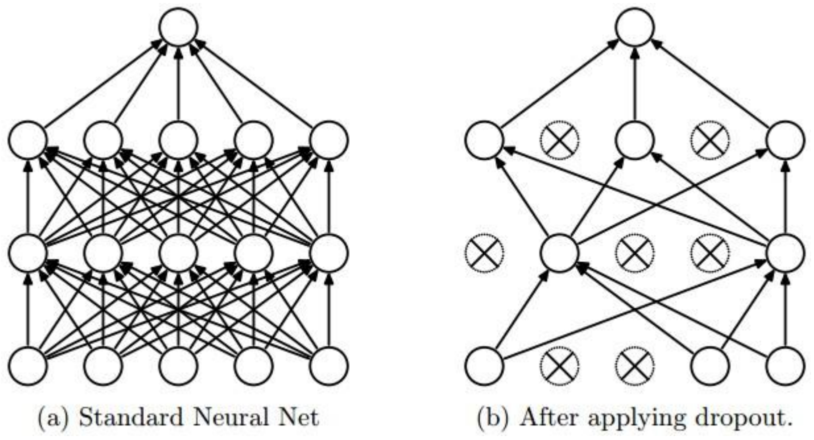
Fully Connected layers are also known as dense layer. Here each hidden layer neuron is connected to every output neuron.
Dropout refers to dropping out units (both hidden and visible) in a neural network. Dropout is a form of regularization. It aims to help prevent overfitting by increasing testing accuracy, for each mini-batch in the training set, dropout layers, with probability p, randomly disconnect inputs from the preceding layer to the next layer in the network architecture. Here, I randomly disconnect with probability p=0.4. After the forward and backward pass are computed for the minibatch, we re-connect the dropped connections, and then sample another set of connections to drop.

Parameters:

- **Units:** 128 (number of neurons in the layer).
- **Activation:** ReLU (Rectified Linear Unit) for introducing non-linearity.
- **Dropout:** Fraction of units to drop during training (e.g., 0.2 before tuning, 0.4 for RNN/LSTM, and 0.2 for GRU after tuning)

Output Shape:

(None, 128): The output is a vector of size 128



3.3.8. Activation layer

Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron. I have used the **softmax** activation function in this project.
The Softmax regression is a form of logistic regression that normalizes an input value into a vector of values that follows a probability distribution whose total sums up to 1. The output values are between the range [0,1] which is nice because we are able to avoid binary classification and accommodate as many classes or dimensions in our neural network model. This is why softmax is sometimes referred to as a multinomial logistic regression. The function is usually used to compute losses that can be expected when training a data set. Known use-cases of softmax regression are in discriminative models such as Cross-Entropy and Noise Contrastive Estimation. These are only two among various techniques that attempt to optimize the current training set to increase the likelihood of predicting the correct word or sentence.

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} , i \in 1,2,3, \dots , k$$

4. MODEL COMPILATION

We used the following for the model compilation: -

4.1. Word2Vector Embeddings

Word2Vector is a neural network-based model that generates dense vector representations of words. The basic idea behind Word2Vec is to train a neural network to predict the context words given a target word, and then use the resulting vector representations to capture the semantic meaning of the words. It captures semantic relationships between words using two main approaches: Continuous Bag of Words (CBOW) and Skip-gram. Although it **c**aptures semantic relationships between words, the main constraint of Word2Vector is that it requires training on a large corpus, doesn’t handle out-of-vocabulary (OOV) words well.

4.2. Adam Optimizer

Adam, is by far the most popular and widely used optimizer in Deep Learning. In most cases, you can blindly choose the Adam optimizer and forget about the optimization alternatives. Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training.
However, in ADAM, a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. The math representation can be simplified in the following way:
$Weights = Weights - (Momentum\ and\ Variance\ combined)$

4.3. Categorical Cross Entropy loss

is a loss function commonly used for multiclass classification problems where each sample belongs to one of several classes (e.g., positive, negative, neutral in your case). It measures the difference between the predicted class probabilities and the actual one-hot encoded labels. The categorical cross entropy loss will be equated as: -

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- $L(y, \hat{y})$ is the categorical cross-entropy loss.
- y_i is the true label (0 or 1 for each class) from the one-hot encoded target vector.
- $\hat{y}_i$ is the predicted probability for class i.
- C is the number of classes.

5. MODEL TRAINING

5.1. Training Parameters

Model trained with 20 epochs with a batch size of 10 before tuning. The models were assigned a learning rate of 0.01 where results were validated with 10% of the dataset.

5.2. Metrics:

- **Accuracy:** Measures the proportion of correctly classified samples.
- **Precision:** Measures how many of the predicted positive instances were actually correct. Defined as Precision = TP / (TP + FP), where TP is True Positives and FP is False Positives.
- **Recall (Sensitivity):** Measures how many actual positive instances were correctly identified. Defined as Recall = TP / (TP + FN), where FN is False Negatives.
- **F1-Score:** The harmonic mean of precision and recall, balancing the two metrics. Defined as F1-Score = 2 * (Precision * Recall) / (Precision + Recall). This is useful when dealing with imbalanced classes.

6. HYPERPARAMETER TUNING

- Hyperparameter tuning is used to optimize model performance by selecting the best combination of parameters. Hyperparameters tuning model Performance by finding the right hyperparameters can significantly improve accuracy, precision, recall, and F1-score. It also prevent overfitting for instance proper tuning of dropout rates and other regularization parameters helps the model generalize better to unseen data.

Method:

Keras Tuner is a library that automates the process of hyperparameter tuning. It uses search algorithms to explore the hyperparameter space and find the best combination.Used Keras Tuner to optimize the following parameters:

- Number of units in RNN/LSTM/GRU layers.
- Dropout rates.
- Choice of optimizer (e.g., Adam, RMSprop).
- Learning rate.
- Batch size

Defination of a range of possible values:

- RNN/LSTM/GRU Units: [64, 128, 256].
- Dropout Rate: [0.2, 0.3, 0.4].
- Optimizer: ['adam', 'rmsprop'].
- Batch Size: [32, 64].
- Learning rate: [0.1, 0.01.0.001]

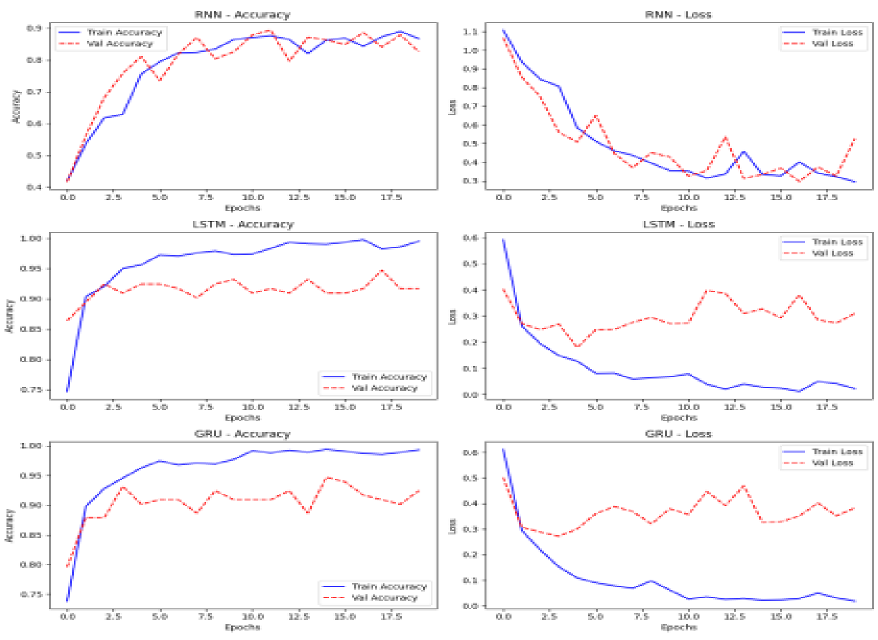
Best Hyperparameters Identified:

- RNN: 128 units, dropout = 0.2, optimizer = RMSprop.
- LSTM: 256 units, dropout = 0.4, optimizer = Adam.
- GRU: 256 units, dropout = 0.4, optimizer = Adam.

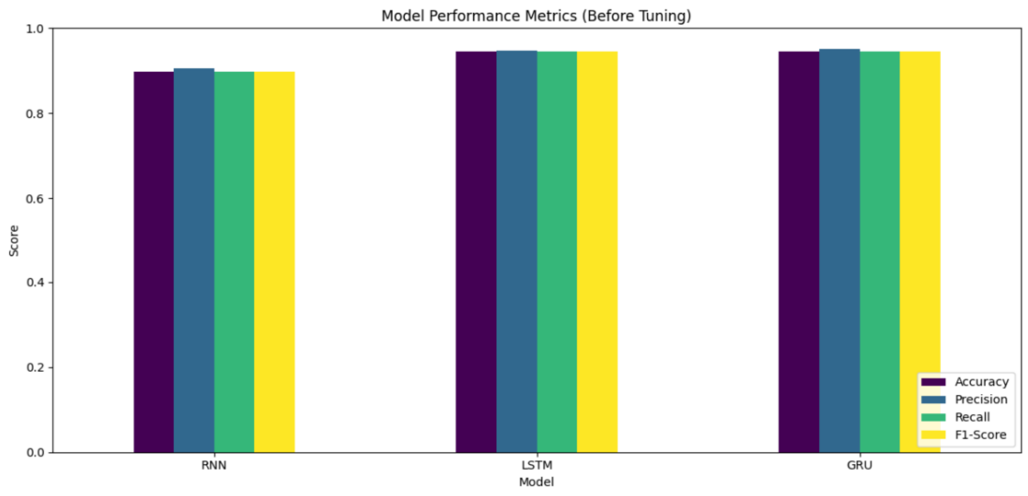
7. RESULT

1. Model performance before hyperparameter tuning

- LSTM and GRU outperformed RNN in terms of accuracy, precision, recall, and F1-score.
- RNN struggled with long-term dependencies, leading to lower performance.
- LSTM achieved the highest accuracy (94.65%), followed closely by GRU (94.52%), while RNN lagged behind (89.72%).

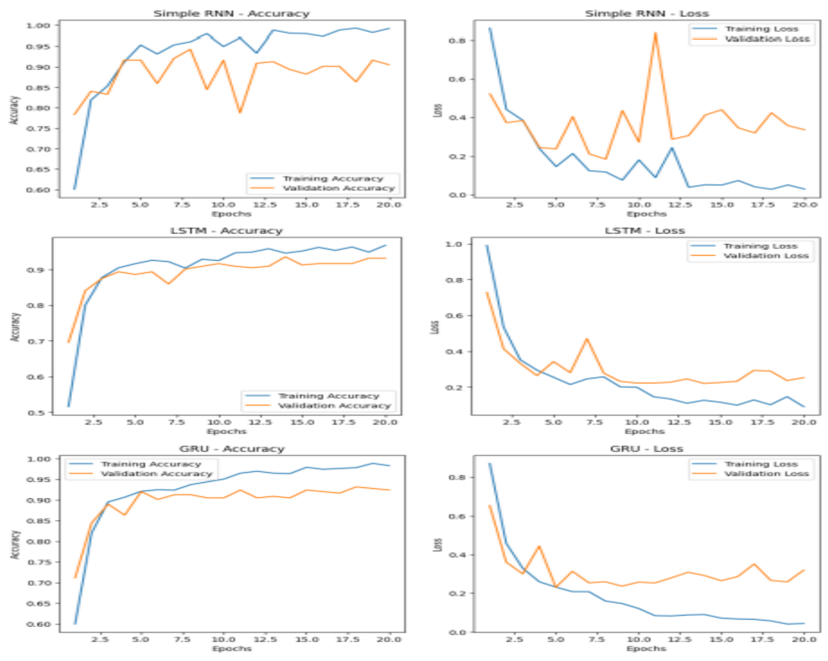


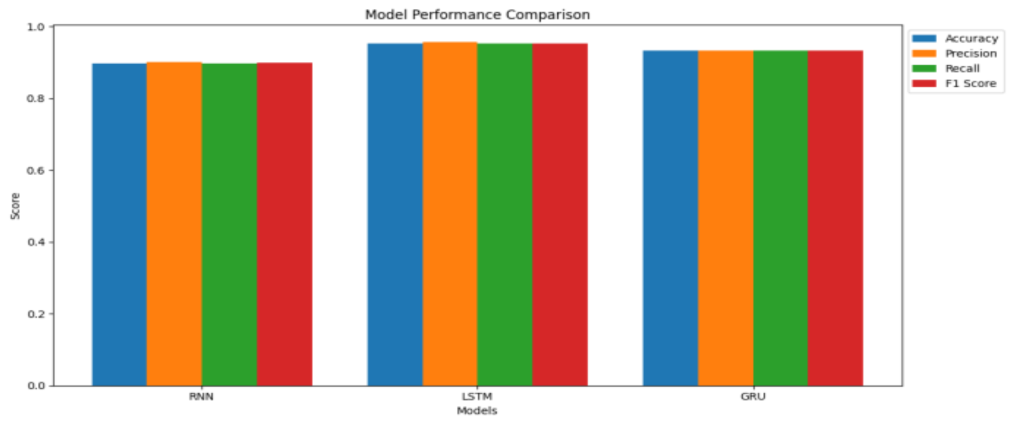
Model performance before hyperparameter tuning



2. Model performance after hyperparameter tuning

- Hyperparameter tuning led to slight improvements in model performance.
- LSTM remained the best-performing model, achieving an accuracy of 95.21%.
- GRU also performed well, with an accuracy of 93.15%, while RNN showed no significant improvement (89.72%).





8. CONCLUSION

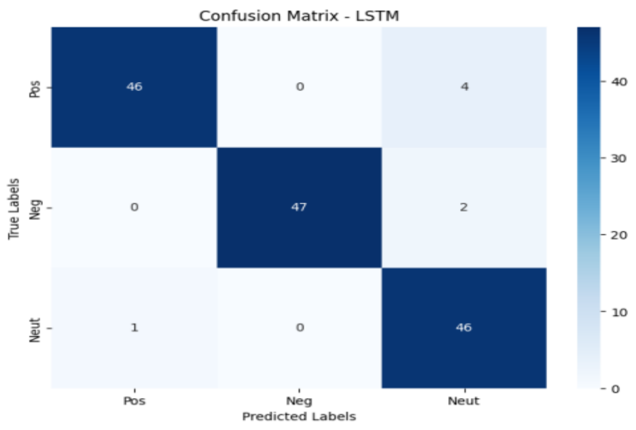
- Generally, after hyperparameter tuning both models slightly improved their performances. STM emerged as the best model for this sentiment analysis task, achieving the highest accuracy (95.21%) after hyperparameter tuning. GRU also performed well with accuracy (93.52%) and could be a good alternative when computational efficiency is a priority. RNN, while simpler, showed limitations in handling long-term dependencies and did not benefit significantly from tuning which resulted accuracy (89.73%)

- LSTM's memory cells and gating mechanisms (input, forget, and output gates) allow it to retain important information over long sequences, making it ideal for sentiment analysis.
  Unlike RNN, LSTM can effectively capture dependencies between words that are far apart in a sentence, which is crucial for understanding sentiment.

- GRU has fewer parameters than LSTM (no output gate), making it faster to train and less prone to overfitting. GRU achieved nearly the same accuracy as LSTM, making it a good choice for tasks where computational resources are limited.

- RNNs struggle with long-term dependencies due to the vanishing gradient problem, which makes it difficult to learn relationships between distant words in a sequence. Even after hyperparameter tuning, RNN's performance did not improve significantly, highlighting its limitations for complex NLP tasks.

LSTM Confusion Matrix showing prediction of true labels.



9. RECOMMENDATIONS
- Use LSTM for sentiment analysis tasks where accuracy is the top priority.
- Consider GRU for scenarios where computational efficiency is important.
- Avoid RNN for tasks involving long sequences or complex dependencies.

10. FUTURE WORK
- Use a larger and more diverse datasets to improve model generalization.
- Explore Transformer-based models (e.g., BERT) for better performance on sentiment analysis tasks.
- Use domain-specific embeddings (e.g., trained on customer reviews) to capture context-specific semantics.

11. REFERENCES

[1] A. Hassan, “SENTIMENT ANALYSIS WITH RECURRENT NEURAL NETWORK AND UNSUPERVISED Ph . D . Candidate : Abdalraouf Hassan , Advisor : Ausif Mahmood Dep of Computer Science and Engineering , University of Bridgeport , CT , 06604 , USA,” no. March, pp. 2–4, 2017.

[2] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation.

Cho, K., et al. (2014). Learning Phrase Representations using RNN EncoderDecoder for Statistical Machine Translation. arXiv preprint arXiv:1406.1078.

[3] Mikolov, T., et al. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv preprint arXiv:1301.3781.

[4] Course Notes