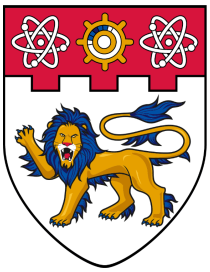

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC4002 Natural Language Processing
Project Report: Group 6

Name	Matriculation Number
Alex Khoo Shien How	U2220791B
Leong Hong Yi	U2120932C
Lim Jun Hern	U2120981B
Ng Yong Jian	U2120431E
Oo Yifei	U2120933E

* Each group member contributed equally to the project.

College of Computing and Data Science
Nanyang Technological University, Singapore

Table of Contents

Part 1. Preparing Word Embeddings

In this assignment, we employed the **GoogleNews-vectors-negative300 Word2Vec pre-trained model** for token embeddings. To ensure consistency, our data preprocessing pipeline for this NLP task includes several key steps: **tokenization, punctuation removal, case folding, and lemmatization**.

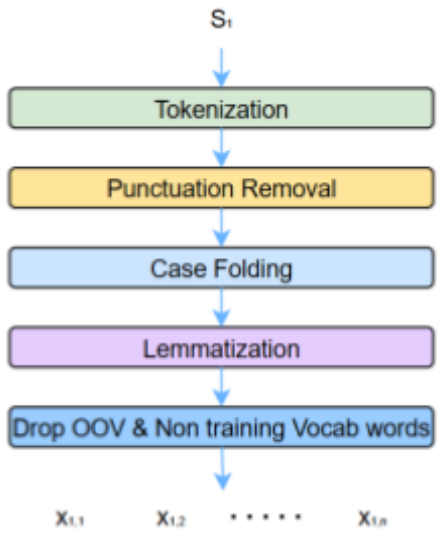

Figure 1.0.1: The final preprocessing strategy pipeline.

Tokenization is the process of breaking down text into individual tokens, which can be words, phrases, or symbols. We utilized **NLTK's WordPunctTokenizer**, which effectively splits sentences while preserving punctuation as separate tokens. This tokenizer is particularly useful because it handles various edge cases, such as contractions and special characters, ensuring that the resulting tokens maintain the structure of the original text.

After **tokenization, punctuation marks are removed** before performing case folding. **Case folding** involves converting all text to lowercase while punctuation removal enables the model to solely focus on the words themselves. These steps are important for uniformity to reduce redundancy and help the model learn more effectively..

We also considered the optional step of removing stopwords from the text by utilizing NLTK's list of common stopwords, such as "the," "is," "in," and "and," which generally carry little meaning on their own. Although removing stopwords is a common practice in many NLP tasks aimed at reducing dimensionality and enhancing model performance, our exploration indicated that this approach did not yield better results for our specific application as negation words in particular serve an important purpose in sentiment negation**.** Consequently, we chose to **retain stopwords** in our preprocessing pipeline, as this allows the model to capture the context and meaning of the text more effectively.

Finally, we performed **lemmatization**, a process that reduces words to their base or root form, known as the lemma. For this, we used NLTK's WordNetLemmatizer, which employs the WordNet lexical database to ensure accurate transformations. This step is vital for consolidating different inflections of a word, thereby enhancing the model's ability to generalize across variations and improving its performance on various NLP tasks.

1a. Training Data Vocab Size

Without handling the OOV, we assessed the vocabulary size of the training dataset and noted a reduction from 18,030 to 16,570 words after preprocessing. This change demonstrates the effectiveness of preprocessing in minimizing redundancies stemming from inflections, plurals, and grammatical variations.

Vocabulary Size: *(From the "SC4002_G6_Vanilla RNN.ipynb" Notebook, more details on this in the README)*

Before Preprocessing: 18,030	**After Preprocessing: 16,570**

1b. Training Data OOV

We also determine the count of out-of-vocabulary (OOV) words, those absent from the vocabulary. The OOV count shows a slight decrease from 3,613 to 3,585 after preprocessing, despite the notable reduction in overall vocabulary size.

OOV Count: *(From the "SC4002_G6_Vanilla RNN.ipynb" Notebook, more details on this in the README)*

Before Preprocessing: 3,613	**After Preprocessing: 3,585**

1c. OOV Handling

Note: For questions before 3(b), words not represented in the pre-trained Word2Vec embeddings or the training set were simply removed from the datasets.

For a more advanced method of representing OOV words in the embedding metric, we experimented with FastText but observed limited improvement, so we opted to **generate embeddings statistically** from the existing Word2Vec vectors. We created random embeddings for OOV words by sampling from a normal distribution with the calculated mean and variance of the pre-trained embeddings. This method provides a statistically consistent representation of OOV words, allowing the model to learn from them while minimizing the impact of missing data. Although these generated embeddings are approximations, they offer effective vector representations of OOV words, enhancing the model's ability to generalize to unseen terms during training.

```python
def create_embedding_matrix(w2v_model, word2index, embedding_dim=300):
    vocab_size = len(word2index)  # Number of words in the training vocabulary

    # Initialize the embedding matrix with zeros (or any other value)
    embedding_matrix = np.zeros((vocab_size + 1, embedding_dim))
    mean = np.mean(w2v_model.vectors, axis=0)
    std = np.std(w2v_model.vectors, axis=0)

    for word, idx in word2index.items():
        if word in w2v_model:
            # If the word exists in the Word2Vec model, use its pre-trained embedding
            embedding_matrix[idx] = w2v_model[word]
        else:
            # Assign embeddings according to random normal distribution with calculated mean and sd
            embedding_matrix[idx] = np.random.normal(
                loc=mean, scale=std, size=(embedding_dim,)
            )

    return embedding_matrix
```

Figure 1.3.1: OOV handling with embeddings generated statistically.

Part 2. Model Training & Evaluation - RNN

RNNs process sequential data by maintaining a hidden state to capture temporal dependencies. For this part, we followed the requirements specified in the project handout, including freezing word embeddings, employing a mini-batch strategy, etc. To leverage the mini-batch strategy, we added **padded tokens** to standardize input lengths across batches, ensuring consistent processing. **PyTorch** was the primary framework used throughout the project.

Considering this is a binary classification task, we employ the **Sigmoid activation function** along with **Binary Cross-Entropy Loss (BCELoss)** function. We also adopted strategies like **dropout layer** to prevent overfitting.

2a. Final Configuration of Best Model

Optuna is an open-source hyperparameter optimization framework that automates the search for optimal configurations in machine learning models, utilizing Bayesian Optimization with a Tree-structured Parzen Estimator (TPE). By using Optuna, we streamline the tuning process and significantly reduce the time required for hyperparameter optimization. This approach systematically fine-tunes parameters, achieving better model performance than traditional manual or grid search methods. Since running Optuna is time-consuming and gives different optimal parameters with each run due to its randomness, we have commented out the tuning code in our notebook. However, it remains included to illustrate our tuning process.

Note: The Optuna hyperparameter tuning strategy will also be applied in the following sections.

As discussed with the professor, while using a multi-head self-attention layer (MHSA) mechanism yielded the highest performance, for the purposes of this report's discussion on vanilla RNN, we focus on commonly used aggregation methods, specifically selecting max pooling as the best alternative. MHSA is thus presented as an innovative approach rather than the primary focus. Further details on MHSA can be found in the response to Question 2(c). Consequently, the "best model" for vanilla RNN in this report refers to the model with max pooling as the aggregation method. The final configuration of our **max-pooling model** is shown in Figure 2.1.1. Note that due to early stopping of patience 5, **the actual training stops at Epoch 14, and the best model is obtained at Epoch 9**.

```
# The Best Hyperparameters
hidden_size = 128  # Hidden size choices
num_layers = 2  # Number of RNN layers choices
learning_rate = 0.01  # Learning rate choices
batch_size = 64  # Batch size choices
num_epochs = 50  # num epoch choices
optimizer_name = 'SGD'  # Optimizer choices
```

Figure 2.1.1: The final configuration of the best model.

2b. Accuracy Score of Validation and Test Set

The test accuracy for the model described in Question 2(a) is 0.7702. The validation accuracy for each epoch during training is displayed in the Table 2.2.1 below.

Table 2.2.1: The validation score for each epoch.

Epoch	Validation Accuracy
1	0.5169
2	0.5994
3	0.6970
4	0.7373
5	0.7617
6	0.7458
7	0.7570
8	0.7674
9 (Best Model)	**0.7824**
10	0.7617
11	0.7233
12	0.7542
13	0.7739
14 (Early Stopping Triggered)	0.7795

2c. Aggregation Strategies

For this section, we explored four different approaches for aggregating RNN outputs. Three of these are more conventional methods: Last Hidden State, Max Pooling, and Mean Pooling. In addition, we experimented with a more innovative approach—using Multi-Head Self-Attention (MHSA) to weight the output before applying mean pooling. Other innovative approaches are introduced in the subsequent sections in Part 3, where they are explained in greater detail.

For each approach, parameters were re-tuned, and the reported accuracy reflects the optimal configuration for that method, as shown in the Table 2.3.1 below. Due to space limitations, specific hyperparameter values for each approach are not detailed here.

Table 2.3.1: The aggregation strategy explored and their accuracies.

	Last Hidden State	**Max Pooling**	Mean Pooling	MHSA - Mean
Description	Uses the final hidden state of the RNN as the representation of the input sequence.	**Selecting the maximum value across all hidden states in the sequence.**	Averages all hidden states in the sequence to create a single representation.	Uses MHSA to weigh sequence features, followed by mean pooling.
Ignore Padded Tokens	False	**True, by masking.**	True, by masking.	True, by masking.
Test Accuracy	0.5807	**0.7702**	0.7664	0.7842

The architecture diagram for each strategy is shown in Figure 2.3.2.

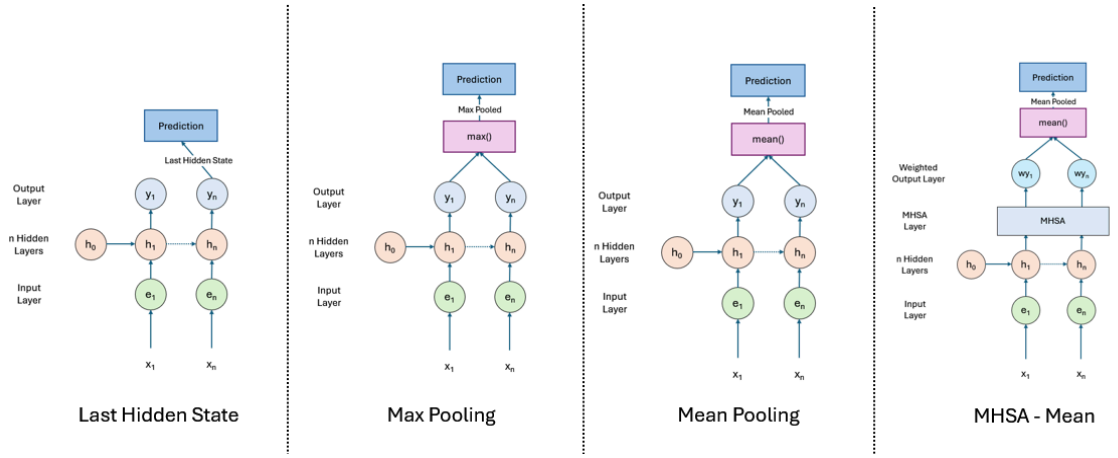

Figure 2.3.1: The architecture diagram for each strategy.
Other layers like the dropout layers are not shown in the diagram.

Although using MHSA increases the test accuracy to 0.7842, as mentioned in the previous section, it is outside the primary scope of our discussion on vanilla RNN. Therefore, we proceed with **max pooling** as the aggregation method for our vanilla RNN model.

Part 3. Enhancement

3a. Unfreeze Word Embeddings

Unfreezing the word embeddings enables gradient flow, allowing the embeddings to be updated during training. This is done by setting *freeze = False* in the model's *nn.Embedding.from_pretrained* component. By making changes based on the best model described in Question 2(a), **this approach achieves 0.7889 test accuracy,** with the best model obtained at Epoch 2**.** Note that the hyperparameters were re-tuned, and the optimal hyperparameter configuration for this setup is shown in Figure 3.1.1.

```
# The Best Hyperparameters
hidden_size = 128  # Hidden size choices
num_layers = 1  # Number of RNN layers choices
learning_rate = 0.001  # Learning rate choices
batch_size = 32  # Batch size choices
num_epochs = 50  # num epoch choices
optimizer_name = 'RMSprop'  # Optimizer choices
```

Figure 3.1.1: The best hyperparameters for Question 3(a).

3b. Proposed OOV Handling Strategy

Here, we adopt the strategy described in Question 1(c) on the model described in Question 3(a) to handle OOV. **This approach achieves a test accuracy of 0.8021,** with the best model obtained at Epoch 1**.** Note that the hyperparameters were re-tuned, and the optimal hyperparameter configuration for this setup is shown in Figure 3.2.1.

```
# The Best Hyperparameters
hidden_size = 256  # Hidden size choices
num_layers = 2  # Number of RNN layers choices
learning_rate = 0.001  # Learning rate choices
batch_size = 32  # Batch size choices
num_epochs = 30  # num epoch choices
optimizer_name = 'Adam'  # Optimizer choices
```

Figure 3.2.1: The best hyperparameters for Question 3(b).

3c. RNN Variations

Both the BiLSTM and BiGRU models use **bidirectional layers** to capture richer context by processing sequences in both forward and backward directions. The forward layers capture context from start to end, while the backward layers work in reverse, ensuring comprehensive representation. The combined outputs from these layers result in a more holistic sentence representation, capturing dependencies that unidirectional models might miss. Their capabilities to handle long-term dependencies are explained in the following sections.

3c(i). Bidirectional Long Short-Term Memory (BiLSTM)

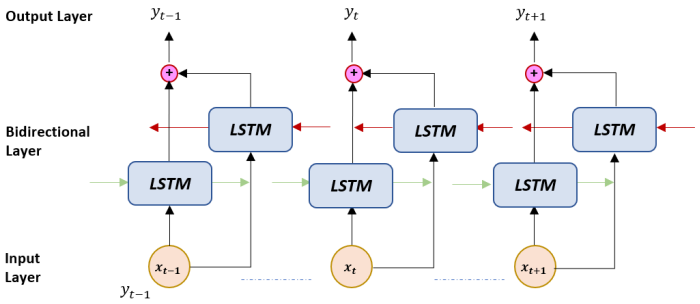

Figure 3.3.1: General model architecture for BiLSTM.

Figure 3.3.1 illustrates the architecture of our BiLSTM model. LSTM handles long-term dependencies through its use of three gates—**forget, input, and output gates**—which regulate the flow of information and retain essential data over long sequences by maintaining a cell state.

Experiments among different aggregation strategies have indicated that **max-pooling strategy** outperformed all other strategies. Further experimentation on advanced mechanisms such as multi-head state attention with max-pooling did not yield better results on test accuracy *(refer to the Appendix)*. We also tested Batch Norm Layers and Residual Connections to enhance model stability and potentially improve generalization.

Table 3.3.1 presents the best configurations and their performances. Our final experimentation using **Batch Norm Layers and Residual Connection with Max Pooling** has outshined all other strategies, with a **test accuracy of 0.7983**.

Table 3.3.1: The best parameters for the BiLSTM model.

Aggregation	**Hidden Size**	**LSTM Layers**	**LR**	**Batch Size**	**Epochs**	**Optimizer**	**Test Acc**
Max Pooling	256	2	0.1	64	60	SGD	0.7974
Max Pooling with Residual Connection and Batch norm	128	2	1e-05	32	50	Adam	0.7983

3c(ii). Bidirectional Gated Recurrent Unit (BiGRU)

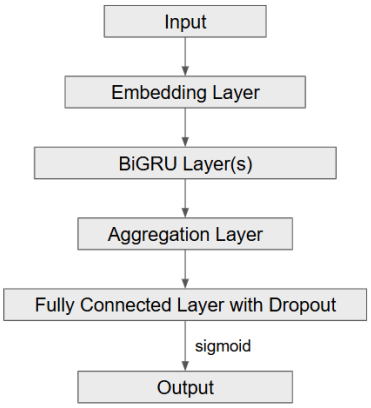

Figure 3.3.2 Architecture of BiGRU.

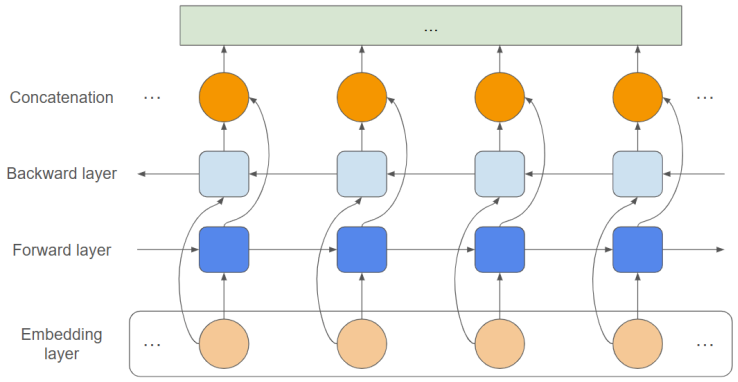

Figure 3.3.3 General model architecture of BiGRU.

In our experiments to replace the RNN with a BiGRU model, Optuna was utilized for hyperparameter optimization tuning to identify the optimal configurations. Figure 3.3.2 and Figure 3.3.3 illustrate the architecture of our BiGRU model. In contrast to LSTMs, GRU simplifies the process of capturing long term dependencies by using only two gates—**update and reset**—which combine the functions of LSTM's gates, reducing the number of parameters required and computational overhead.

We compared among four aggregation strategies: max pooling, mean pooling, max-mean pooling, and MHSA-mean pooling. We also compared among four different optimizers: Adam, SGD, RMSprop, and AdamW. Other configurations were hidden size, number of GRU layers, learning rate, batch size, epochs, and whether to ignore the padded tokens in the aggregation step.

Table 3.3.2 presents the best configuration and its performance. Overall, all strategies achieved accuracies above 0.8 on both validation and testing sets, with **max-mean pooling yielding the highest testing accuracy of 0.8208**. For performance details of other configurations, please *refer to the Appendix*.

Table 3.3.2: The best parameters for the BiGRU model.

Aggregation	Hidden Size	GRU Layers	LR	Batch Size	Epochs	Optimizer	Test Acc
Max Pooling	256	2	0.001	32	20	RMSprop	0.8208

3d. Convolutional Neural Network (CNN)

In our experiments with replacing the RNN with a CNN model, we leveraged Optuna for hyperparameter tuning to determine the optimal settings for the experiments. The specifics of how the filters are convolved through the embeddings are depicted in Figure 3.4.1. This figure illustrates a filter size of 3 applied to the embeddings, enabling the model to capture various features in the text. **By sliding the filter across word embeddings, the CNN effectively extracts local patterns**, which enhances the model's ability to identify meaningful features.

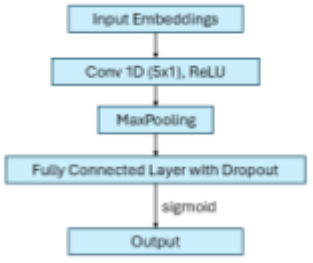

Figure 3.4.2: Architecture of CNN.

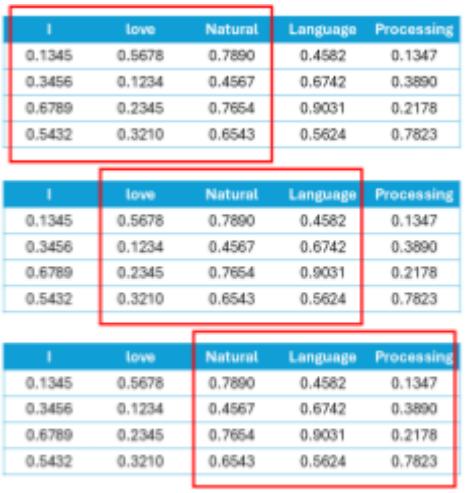

Figure 3.4.1: Convolution of filters of size 3 through word embeddings.

Additionally, we compared two aggregation strategies in the CNN: **max pooling and mean pooling**. We used pooling across the word sequence to capture the features of the convolutional filters. Validation results indicate that **max pooling outperformed mean pooling**, achieving a validation accuracy of **0.8021** compared to **0.7927** with mean pooling.

We also **experimented with adding an attention head after max pooling**. However, this addition showed no significant improvement.

Using our **best CNN model with max pooling** as the aggregation strategy and no attention layer, we achieved a **final test accuracy of 0.7852**. The model architecture is illustrated in Figure 3.4.2. The optimal hyperparameters can be found in Table 3.4.1.

Table 3.4.1: The best parameters for the CNN model.

Number of filters	256
Number of Convolution Layers	1
Filter size	7
Batch Size	32
Learning Rate	0.0005
Optimizer	Adam
Test Accuracy	0.7852

3e. Final Improvement Strategy - Transfer Learning

Several approaches exist for enhancing model performance, such as collecting more data, POS tagging, and negation scope detection. However, following the professor's suggestion to focus on a single method, we only implemented **transfer learning using a pre-trained Transformer model**, as this approach is likely to give the best performance.

Transformers leverages advanced deep learning techniques to capture linguistic patterns and relationships, providing advantages over traditional methods in terms of efficiency and accuracy. Additionally, transfer learning is a technique in machine learning that enables the efficient adaptation of the pre-trained models to specific tasks. This approach significantly reduces the need for task-specific training data while maintaining high performance through fine-tuning.

BERT is a transformer-based neural network architecture in the field of NLP. For this project, we experimented with both the BERT base model and RoBERTa, both derived from the BERT Transformer. Since RoBERTa is particularly tailored for sentiment analysis and achieved higher test accuracy in our project, this section will primarily focus on the RoBERTa model, specifically the *'roberta-base'* variant, due to page limit.

To perform transfer learning with RoBERTa, we first need to tokenize our inputs in a format compatible with the model. Unlike traditional neural networks, RoBERTa does not require lemmatization or other preprocessing techniques. The model is able to analyze all contextual information in a sentence, including punctuations, from multiple perspectives. Thus, we apply the RoBERTa tokenizer directly to our inputs, utilizing the generated input IDs and attention masks for subsequent training.

To mitigate overfitting, we include a **dropout layer**, which randomly deactivates certain neurons during training. Following this, a **linear layer** is used to adapt the RoBERTa output to our sentiment classification task. Finally, a **sigmoid activation** function is applied to produce probabilities, allowing the model to output a value between 0 and 1. The class definition is shown in Figure 3.5.1.

```python
class BERTRoBERTa(nn.Module):
    def __init__(self):
        super(BERTRoBERTa, self).__init__()
        self.bert = RobertaModel.from_pretrained(PRE_TRAINED_MODEL_NAME, return_dict=False)
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(self.bert.config.hidden_size, 1)  # Output size is 1 for binary classification

    def forward(self, input_ids, attention_mask):
        _, pooled_output = self.bert(input_ids=input_ids, attention_mask=attention_mask)

        output1 = self.dropout(pooled_output) # Apply dropout
        output2 = self.fc(output1) # Final fully connected layer

        # Sigmoid activation for binary classification
        output3 = torch.sigmoid(output2)

        return output3
```

Figure 3.5.1: Class definition of the RoBERTa-based classifier.

For optimization, we employ the **AdamW optimizer**, which effectively handles weight decay in transformer-based models and helps prevent overfitting. A **linear learning rate scheduler with warm-up** is implemented to gradually increase the learning rate during initial training steps, followed by a controlled decrease. We also applied **gradient clipping**. These approaches enhance model stability and allow for better adaptation to the task.

Before training the model, we **freeze the earlier layers** up to *'bert.encoder.layer.9.attention.self.query.weight'*. This selective layer freezing leverages transfer learning by retaining the general language features in the earlier layers while allowing task-specific adaptations in the subsequent layers, which helps mitigate overfitting. We did not unfreeze more layers to optimize computational resources and maintain model performance.

While designing the model, we also experimented with adding an additional multi-head self-attention (MHSA) layer and a residual connection (to improve gradient flow) with normalization to the RoBERTa output, in an attempt to enhance the model's ability to capture complex dependencies and improve feature representation. However, these modifications did not significantly improve accuracy. Therefore, **we retained the architecture shown in Figure 3.5.1 as our final model**. A comparison of two different models is illustrated in Figure 3.5.2.

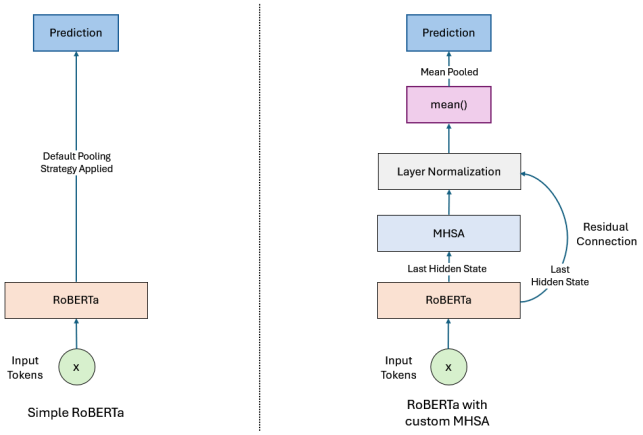

Figure 3.5.2: A simplified diagram showing the comparison of two different architectures. The relevant codes of using MHSA are modified and commented out in our python notebook, so it may not accurately reflect the structure illustrated in the diagram.

The other best hyperparameters for RoBERTa is as follow:

Table 3.5.1: Other best hyperparameters for RoBERTa.

Batch Size	128
Early Stopping Patience	5
Learning Rate	1e-05
Number of Epochs	Max: 50, Best Model Obtained At: 12

The test accuracy of the trained model after applying transfer learning on RoBERTa is 0.8874. In contrast, the test accuracy of the trained model after applying transfer learning on the BERT base model is 0.8462. This demonstrates that RoBERTa performs better on the required tasks in the project.

Thus, **our final improvement strategy is performing transfer learning on RoBERTa** with the techniques described above.

3f. Discussions

The exact implementation of the strategies will not be repeated. Please refer to the earlier sections for more details.

Qn.	Description	Test Acc.	Discussion
2(c)	Vanilla RNN - Last Hidden State	0.5807	Using only the last hidden state yielded the lowest performance, likely because it captures only the **final time step's information**, which can lead to loss of contextual details from earlier in the sequence.
2(c)	Vanilla RNN - Max Pooling	0.7702	Both max and mean pooling significantly improved the score over the last hidden state. Max pooling captures the **most prominent features** across the sequence, providing a more balanced summary of the entire sequence. It is also less sensitive to noise and padding, which may reduce sentiment strength by capturing only the strongest activations.
2(c)	Vanilla RNN - Mean Pooling	0.7664	Both max and mean pooling significantly improved the score over the last hidden state. Mean pooling tends to give a **smoother, more holistic summary**. Although key features emphasized by max pooling have marginally higher relevance than a generalized summary from mean pooling, the small difference in scores also indicates that both techniques effectively capture useful information for the model.
2(c)	Vanilla RNN - MHSA Mean	0.7842	MHSA further improved performance by allowing the model to **attend to different parts of the sequence simultaneously**. This mechanism helps capture more nuanced dependencies and contextual information, which standard pooling techniques may miss.
3(a)	Vanilla RNN - Unfreeze Embedding	0.7889	Unfreezing the embeddings further improves the score, as it allows the model to fine-tune word embeddings during training, **aligning them more closely with the task at hand**. This flexibility enhances the model's ability to learn meaningful representations, which boosts overall performance compared to frozen embeddings.
3(b)	Vanilla RNN - Handle OOV	0.8021	Retaining OOV words by assigning them statistically randomly initialized embeddings allows the model to still **process these words even if their meanings aren't fully captured**. This approach helps preserve sentence structure and ensures that input words are not ignored. **Over time, the random embeddings can adapt**, capturing some relationships or positioning in the embedding space, which may be helpful for prediction. This also demonstrates the effectiveness of our statistical random initialization approach.
3(c)	BiLSTM - Max Pooling with Residual Connection and Batch Norm	0.7983	The bidirectional structure, memory cells and gates of the BiLSTM model can capture more **complex and long-term dependencies** that might not be captured effectively by a vanilla RNN. Furthermore, the addition of Residual Connection and Batch Norms layers can effectively **further resolve the vanishing gradient issue** in vanilla RNN and **improve model generalization**. However, the performance remains similar to that of the Vanilla RNN. One possible explanation is that LSTMs have more parameters due to their complex gating architecture, which can lead to **overfitting on small datasets**.

3(c)	BiGRU - Max-Mean Pooling	0.8208	Using a BiGRU offers additional benefits. While providing similar advantages to BiLSTM, such as bidirectionality and the ability to capture long-term dependencies, GRUs are known to be more efficient than LSTMs due to their **simpler gating mechanism**, which can lead to faster convergence and improved performance. Furthermore, the combination of Max-Mean pooling enhances model performance by **capturing both the most significant features and a general overview of the data**. This balanced feature representation leverages the strengths of both max and mean pooling, making the model more robust to variability and improving its generalization capability.
3(d)	CNN - Max Pooling	0.7852	CNNs showed decent performance in sentiment analysis by capturing local patterns (e.g., "very good") through convolutional filters and pooling, focusing on sentiments regardless of position. However, it **struggles to capture long-range dependencies and context**, unlike RNNs, which are crucial for understanding overall sentiment. As a result, CNNs achieved lower test accuracy compared to most RNNs.
3(e)	Transfer Learning on BERT	0.8462	Using BERT for transfer learning provides a significant performance boost over the earlier models. BERT's bi-directional attention mechanism captures rich contextual information from both directions of a sentence, allowing it to **understand complex linguistic patterns**. This improves the model's ability to represent nuanced meanings, leading to a higher score.
3(e)	Transfer Learning on RoBERTa	0.8874	RoBERTa achieves an even higher score. This improvement can be attributed to **several enhancements over BERT**, including longer training on more data, dynamic masking, and larger batches during pre-training. These modifications make RoBERTa's embeddings more expressive and robust, capturing subtle contextual cues more effectively than BERT.

In this project, we explored sentiment analysis using deep learning techniques, focusing on RNNs and enhancements such as word embeddings, OOV handling strategies, and model variations. By experimenting with BiLSTM, BiGRU, we aimed to improve the accuracy and generalizability of sentiment classification. While RNN is commonly used for sequential data processing, CNNs have decent performance by capturing local features within text through convolutional filters. Overall, max pooling tends to perform better as an aggregation method, and this trend is consistent across most of the models.

We also examined transfer learning and used the RoBERTa model as a final enhancement strategy, which offered additional performance gains by leveraging pre-trained embeddings. Through rigorous evaluation, our optimized model demonstrated promising results, indicating that carefully tuned and designed architectures significantly improve sentiment analysis outcomes.

Interestingly, using the Last Hidden State as an aggregation method led to slower convergence compared to pooling methods, possibly due to increased complexity in backpropagation. Additionally, although more complex architectures tend to yield higher performance, they usually require longer training times, especially for our model in Section 3(e). Given the rapid advancements in LLMs and Gen-AI, other techniques like Zero-Shot Learning via Prompt Engineering present valuable avenues for future exploration, as they require minimal training (or, more precisely, Prompt Engineering) and are faster to deploy, while still offering competitive performance.

This project underscores the potential of deep learning in natural language processing tasks and highlights future directions for optimizing language models further.

Appendix

Table A: BiLSTM Experiment Trials Results with Optimal Parameters

Aggregation	Hidden Size	BiLSTM Layers	Learning Rate	Batch Size	Epochs	Optimizer	Val Acc	Test Acc
Max Pooling	256	2	0.1	64	60	SGD	0.7777	0.7974
Mean Pooling	128	3	1e-05	128	50	RMSprop	0.7889	0.7730
Last Hidden State	128	2	0.001	64	50	Adam	0.7683	0.7883
Max-Mean Pooling	256	2	0.001	32	40	RMSprop	0.7749	0.7852
MHSA with Max Pooling	128	2	1e-05	32	50	Adam	0.7674	0.7608
MHSA with Max-Mean Pooling	128	3	0.0001	128	40	RMSprop	0.7771	0.7795
Residual Connection and Batch Norm	Hidden Size	BiLSTM Layers	Learning Rate	Batch Size	Epochs	Optimizer	Val Acc	Test Acc
Max Pooling	**128**	**2**	**1e-05**	**32**	**50**	**Adam**	**0.7627**	**0.7983**
Max-Mean Pooling	128	3	0.0001	128	40	RMSprop	0.7771	0.7795

The mean values in the max-mean pooling strategy may have introduced more noise as compared to max-pooling strategy, causing the strategy to yield suboptimal results for the BiLSTM model. However, it is essential to note the strategy has also produced superior results for the BiGRU model and the explanation can be found in Section 3f: Discussion.

Table B: BiGRU Experiment Trials Results with Optimal Parameters

	Max Pooling	Mean Pooling	**Max-Mean Pooling**	MHSA - Mean
Ignore Padded Tokens	True, by masking	False	**True, by masking**	False
Hidden Size	128	128	**256**	256
GRU Layers	3	1	**2**	1
Learning Rate	0.001	0.001	**0.001**	0.0001
Batch Size	32	64	**32**	32
Epochs	50	40	**20**	40
Optimizer	Adam	RMSprop	**RMSprop**	RMSprop
Val Accuracy	0.8021	0.8011	**0.8077**	0.8124
Test Accuracy	0.8171	0.8021	**0.8208**	0.8002