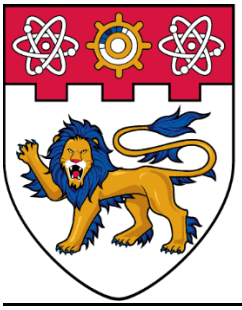


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Semester 1 AY 24/25

SC4002 Natural Language Processing Group Assignment

Group 13: SC4002_G13

Group Members:

Ng Tze Kean (U2121193J)

Ng Woon Yee (U2120622C)

Yap Shen Hwei (U2140630E)

Lim Boon Hian (U2120791F)

Phee Kian Ann (U2122217L)

Lim Ke En (U2122069L)

Contributions	Name
Part 1	Ng Tze Kean
Part 2	Yap Shen Hwei Lim Boon Hian
Part 3(a) & 3(b)	Ng Tze Kean
Part 3(c) BiLSTM	Phee Kian Ann
Part 3(c) BiGRU	Lim Ke En
Part 3(d) CNN	Ng Woon Yee
Part 3(e) & 3(f)	Ng Tze Kean
Report and Documentation	Ng Tze Kean Yap Shen Hwei Lim Boon Hian Ng Woon Yee Phee Kian Ann Lim Ke En

Part 1. Preparing Word Embeddings

Part 1 Discussion

In this part, we prepare the word embeddings for the downstream tasks in the later part. Before we obtain the word embeddings, we are tasked to process the training text, mainly to find out the vocabulary size and out-of-vocabulary (OOV) words.

To start off, we first used a simple regex approach from GPT2 to extract the tokens from the text. We then compared the tokenization process with the NLTK tokenizer and found that the NLTK tokenizer is more accurate in tokenizing the text.

With that in mind, we decided to use the NLTK tokenizer to tokenize the text. Next, we opted to use GloVe as the word embeddings for this task. A simple extraction of the vocabulary size of GloVe is easily done though iterating and extracting the keys of the embeddings.

To obtain the OOV words, we compared the vocabulary of the training data with the vocabulary of GloVe through a set difference operation. We defined several functions to load and tokenize the training set which will be used subsequently in the other parts. To conclude part 1, we inherit the embeddings from GloVe and first ignore the OOV words as they will be subsequently handled in part 3.

Part 1 Answer

(a) Vocabulary Size: The size of the vocabulary is **18030**, after removing the OOV words from Glove embeddings, the final size of the vocabulary is **16163**.

(b) Out-of-Vocabulary (OOV) Words: We identified **1867** OOV words in our training data. These words were present in the training data but not found in the GloVe dictionary

(c) Mitigating OOV Limitations: To mitigate the issue of OOV we first considered the use of `<UNK>` tokens to represent all OOV words. However, this approach may not be ideal as it does not capture the semantics of the OOV words. Instead, we decided to use character-level embeddings to represent OOV words, this is seen in FastText where each word is broken down into character n-grams. This approach allows us to capture valid words that might not be in the vocabulary.

We showcase the snippet of the `<UNK>` token code below:

```
# mapping of words to indices and vice versa
word2idx = {word: idx for idx, word in enumerate(sorted(extended_vocab))}
idx2word = {idx: word for word, idx in word2idx.items()}

print("Building embedding matrix...")
vocab_size = len(word2idx)
print(f"Vocab size: {vocab_size}")
embedding_matrix = np.zeros((vocab_size, EMBEDDING_DIM))

for word, idx in word2idx.items():
    embedding_matrix[idx] = glove_dict[word]

# add random vector for unknown words
embedding_matrix[word2idx[UNK_TOKEN]] = np.random.normal(scale=0.6,
size=(EMBEDDING_DIM,))
```

Part 2. Model Training & Evaluation - RNN

Part 2(a) Answer

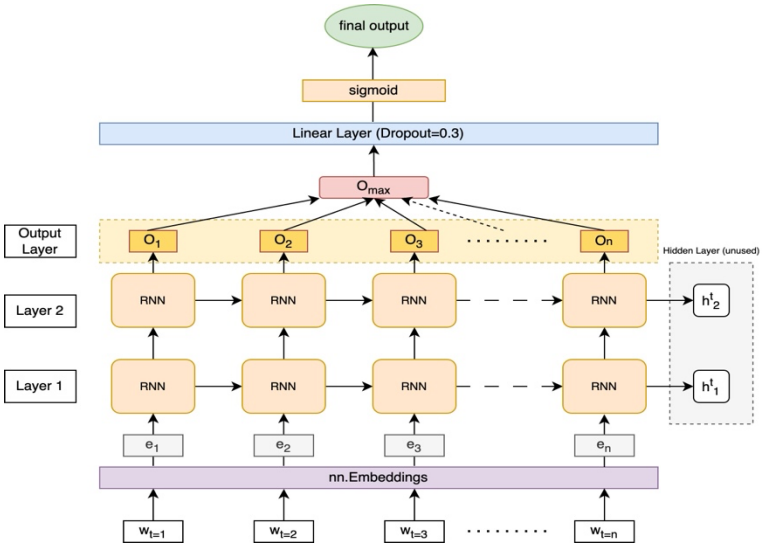


Figure 1

Number of Training Epochs	Initial Learning Rate	Optimizer	Batch Size
14 (early stopped)	0.0001	Adam	32

Table 1

Fixed Hyperparameters	Other Configurations
Dropout at Linear Layer = 0.3, Hidden Size = 128, Num of RNN layers = 2	Early Stopping with patience = 3

Table 2

Part 2(b) Answer

Test Accuracy: 0.773, Validation Accuracies at each epoch during training

Epoch	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
1	0.691	0.68	0.521	0.634
2	0.672	0.654	0.611	0.691
3	0.63	0.594	0.655	0.698
4	0.588	0.568	0.69	0.722
5	0.571	0.555	0.699	0.731
6	0.559	0.552	0.71	0.733
7	0.552	0.541	0.717	0.742
8	0.544	0.538	0.722	0.739
9	0.536	0.529	0.729	0.742
10	0.529	0.524	0.733	0.749
11	0.522	0.517	0.740	0.749
12	0.519	0.513	0.743	0.749
13	0.513	0.51	0.744	0.753
14	**0.508**	**0.511**	**0.750**	**0.754**

Table 3

Part 2(c) Answer

In our final, optimal RNN model, the final sentence representation is a max-pooling of the output layer which is the maximum of all hidden states produced at the output layer (hidden states at every time step), as indicated in *Figure 1*.

The word embeddings fed into the model were also pre-processed as it produced better results.

We had explored the following configurations: Padding with Fixed vs Variable Lengths, Hidden State vs Output Layer, Sorted vs Unsorted Embeddings Dataset.

During our exploration, we had also tested out Gradient Clipping, which we briefly mention in Part 2 Discussion. In training, we removed all OOV words from our dataset.

Padding: Fixed Length or Variable Length

As we are performing mini-batch processing, at each batch, the length of sequence of each input has to be the same. To tackle the issue of different sequence lengths, we used `<PAD>` token, which has a zero-tensor embedding to pad inputs that are shorter than the longest input. Both approaches are summarised in Table 3 below.

Approach	Description	Test Accuracy
Fixed length	Restrict each sentence to 20 tokens by truncating longer sentences or padding the shorter one.	0.553
Variable length	Allow sentences to keep their original lengths and apply dynamic padding during batching using `torch.pad_sequence`. This ensures entire sentences are passed to the model without truncation.	0.730

Table 4

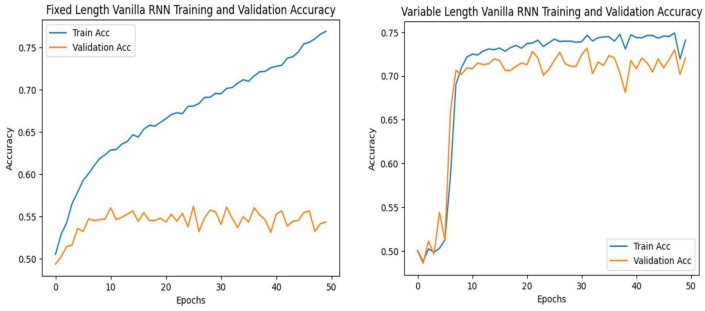


Figure 2

We notice that using `pad_sequence` returns a better result. We believed this is since the model processes the entire sentence instead of truncating the potential important information.

Hidden State vs Output Layer as Input into Linear FC Layer

There are 2 ways that we can configure our input into the Linear FC layer. To see a clearer picture, please refer to *Figure 1.* We performed Max Pooling, Sum Pooling and Mean Pooling on both the Hidden and Output Layers before inputting into the Linear FC Layer.

The first method involves using the Hidden State Layer, which encodes the output of the last time step of each stacked RNN layer (h_1^t, h_2^t). The second method involves using the Output Layer $(o_1, o_2, o_3, \dots, o_t)$ as input into Linear FC layer. The results are summarised in the following tables.

Hidden Layer	Pooling Config	Test Acc
	$Max(h_1^t, h_2^t)$	0.745
	$Sum(h_1^t, h_2^t)$	0.720
	$Mean(h_1^t, h_2^t)$	0.738
Using Last Hidden/ Output state	No Pooling, using $o_n = h_n^t$	0.750

Table 5

Output Layer	Pooling Config	Test Acc
	$Max(o_1, o_2, \dots, o_n)$	0.761
	$Sum(o_1, o_2, \dots, o_n)$	0.736
	$Mean(o_1, o_2, \dots, o_n)$	0.725

Table 6

The results show that in both cases, **Max pooling** achieves better results compared to Sum and Mean pooling. **Max pooling on output layer** achieves the best result. This could be because max pooling picks the most significant word of the sentences, which significantly determines the sentiment of the overall sentence. Max pooling on output layer outperforms Max pooling on hidden layer, possibly also because only 2 stacked RNN layers were used, which meant we only had 2 hidden states from the hidden layer to perform pooling, which may not be the best choice of "representation" when utilising the hidden layer.

Sorted dataset input vs Unsorted dataset input

We tried sorting the train dataset before training our max pool RNN, and this produced a smoother training curve and improved test accuracy (0.761 → 0.773). We believe this is as the model now learns how to identify the key features gradually from shorter sentences to the longer ones.

Part 2 Discussion

Gradient Clipping vs. No Gradient Clipping

We applied gradient clipping to prevent potential bad updates. In vanilla RNN with hidden state, we saw slight improvement in test accuracy from 0.730 to 0.749. Train vs validate accuracy graphs shows slight improvement.

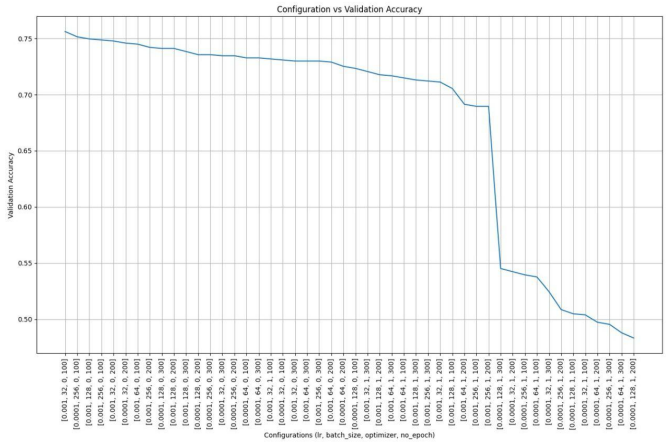
Figure 5

Hyperparameter Search: Grid Search

First, we introduced a dropout rate of 0.3 at the Fully Connected Linear Layer to reduce the chances of overfitting. We then performed a plain grid hyperparameter search on the different permutations of Number of Training Epochs, Type of Optimizer, Initial Learning Rate and Batch Size, with the following fixed hyperparameters: Dropout, Hidden Size, and Number of Layers. We have also implemented Gradient Clipping and Sorted Dataset when training the model.

Number of Training Epochs	Optimizer	Initial Learning Rate	Batch Size	Fixed hyperparameters
100, 200, 300, 400	Adam (0), SGD (1)	0.0001, 0.001	32, 64, 128, 256, 512	dropout=0.3, hidden_size=128, num_layers = 2

Table 5

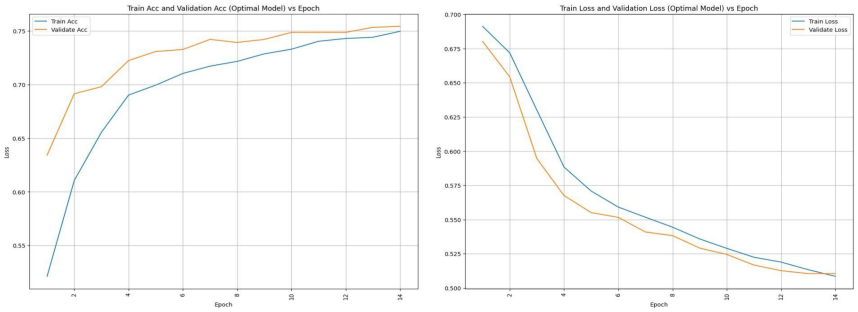


Note: the graph does not show at which epoch the model early stops, if any.

There is a log file generated that allowed us to identify the early stopped epoch.

Figure 6

The best performed model had the following configurations below (Figure 8)**.** The validation curves following the train curves indicate a sign that the model is learning well.



Parameters:
initial_lr = 0.001,
batch_size = 32,
optimizer = 0 (Adam),
num_epochs = 100
early stopping = 14

Figure 7

In our grid search, smaller batch sizes led to higher validation accuracy, likely because smaller batch sizes allow the model to update weights more frequently, so the model captures data patterns more effectively.

Furthermore, Adam outperformed SGD, likely due to its adaptive learning rates and momentum, helping the model converge faster and avoid local minima, leading to better overall accuracy.

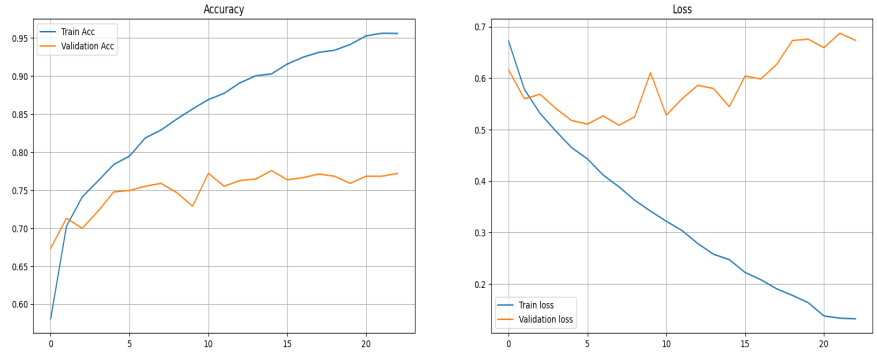
Part 3. Enhancement

Part 3(a) Discussion

As stated by the task, we unfreeze the word embeddings of the RNN model, allowing for updates as the model trains. This is done by passing a parameter to the RNN model to set `freeze` to **`False`**. We then train the model with the train data and evaluate the model with the test data to obtain the accuracy and loss of the model.

Surprisingly, there was degradation of the performance of the model with unfreezed weights even though we expected a more specialised model to handle the task. We hypothesise that the test set contains more generalised schematics of English language and having the loss of generalised embeddings caused the model to be unable to generalise well (overfitting to the train set).

Part 3(a) Answer



Test accuracy: 0.783

Figure 8

Part 3(b) Discussion

We apply the mitigation of OOV words by first using the `<UNK>` token to represent all OOV words. Tokens from the train data that are in the GloVe dictionary will have its embedding taken, and those that are not will be removed from the vocabulary allowing it to be implicitly represented by the `<UNK>` token. The`<UNK>` token will then be assigned a random embedding vector of the same dimension as the GloVe embeddings.

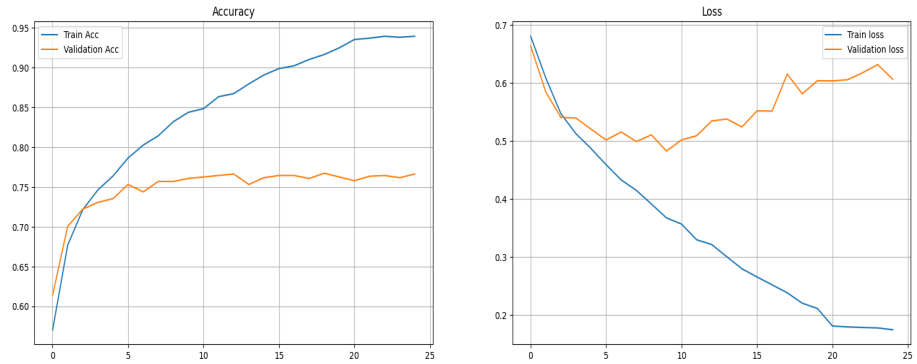
We also have to modify the model to handle OOV words. This is done in the forward pass of the model where we check if the word is in the vocabulary. If it is not, we will use the `<UNK>` token to represent the word. The purpose of this additional step is to ensure that the model is robust enough to handle sequences that are not first processed by the test data loader, such as directly inputting our own text to the model for testing purposes.

Next, we try to implement the FastText model as the embedding layer to replace the GloVe embeddings. To aid us in the implementation, we use the `gensim` library to load the FastText model and then we train the model again with the unfreezed embedding to obtain the accuracy and loss of the model.

To summarise the findings, the `word2vec` weights are taken from a well generalised model while `FastText` is trained on the small dataset, yet its performance is notable, achieving a score close to the `word2vec` model when OOV handling is done. What's interesting is the smoothness of the curve and the rate at which the model's accuracy starts to increase is much sharper for embeddings from `FastText` compared to `word2vec`.

Part 3(b) Answer

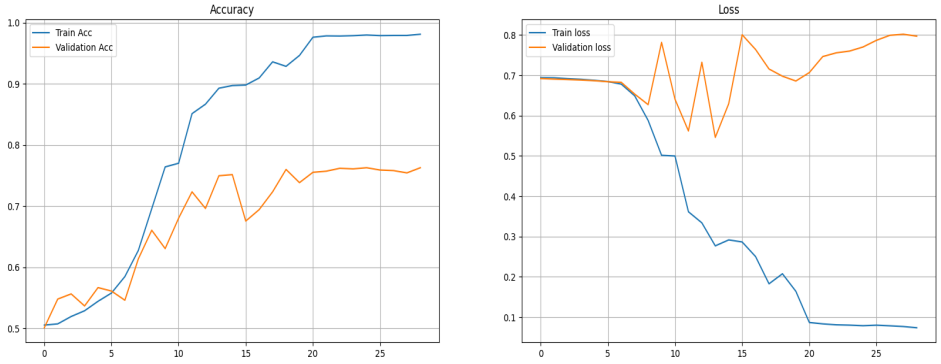
With the inclusion of the `<UNK>` token for handling OOV words, the accuracy and loss is as shown below.



Parameters same as optimal RNN
Test accuracy: 0.799

Figure 9

With the FastText model's embedding the results are as follows



Parameters same as optimal RNN
Test accuracy: 0.780

Figure 10

Part 3(c) Discussion to BiLSTM and BiGRU

BiLSTM

BiLSTM or Bidirectional Long Short-Term Memory, aims to handle the vanishing gradient problem faced by typical RNNs by allowing information to persist over its hidden states through the usage of LSTM unit cells that comprises three components, Forget gate, Input gate and Output gate. The bidirectionality also allows for the network to learn more information from both the past and future states, but at the cost of doubling the model complexity.

Firstly, a naive and simple 2-layer BiLSTM was implemented while following some of the techniques mentioned in the previous sections. Dropout and max pooling were done after passing the pre-trained `<UNK>` token embeddings through the LSTM layer, as max pooling scored the best and dropout is there to mitigate the overfitting issue of LSTMs. It is then passed into a linear FC layer to produce the final output. Gradient clipping was also subsequently done during training to help prevent the exploding gradient problem in RNN/ LSTMs.

Number of Training Epochs	Validation Loss	Validation Accuracy	Test Loss	Test Accuracy
30 (early stopped)	0.517	0.753	0.516	0.771

Table 6

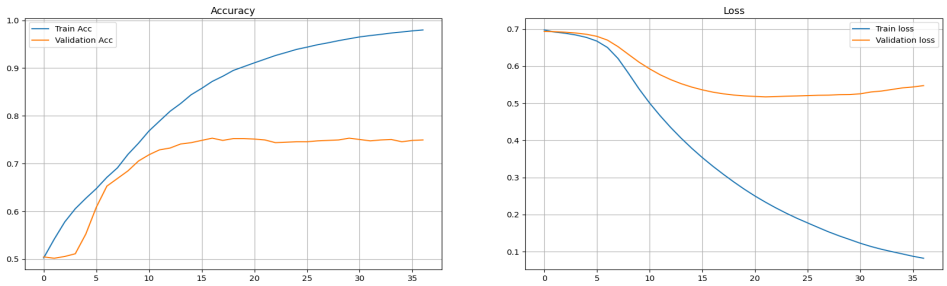


Figure 11

In theory, the LSTM model should be able to perform well on text analysis tasks due to its increased learning of global contextual information, however, LSTMs are very prone to overfitting, especially in a small dataset such as this, which is problematic. This can be seen as the model quickly overfits when increasing the hidden dimension's parameter higher than 8, where the training accuracy quickly hits 90% in less than 10 epochs and the validation loss quickly starts to stagnate around 5 epochs for those scenarios.

BiGRU

BiGRU model stands for Bidirectional Gated Recurrent Unit, which is a type of recurrent neural network (RNN). Similarly to LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time. However, in GRU, the memory cell state is replaced with a "candidate activation vector", which is updated using two gates: the reset gate and update gate. It combines the forget and input gates into a single "update gate." The reset gate determines how much of the previous hidden state to forget while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state.

In this implementation, Spatial dropout is used. The spatial dropout will zeroes out the entire 1D feature map from the embedding feature vector of each word, helping in regularisation.

After the GRU layer, we will concatenate both, the average pooling and max pooling of the hidden representation and the last hidden state of GRU to prevent our model from forgetting information.

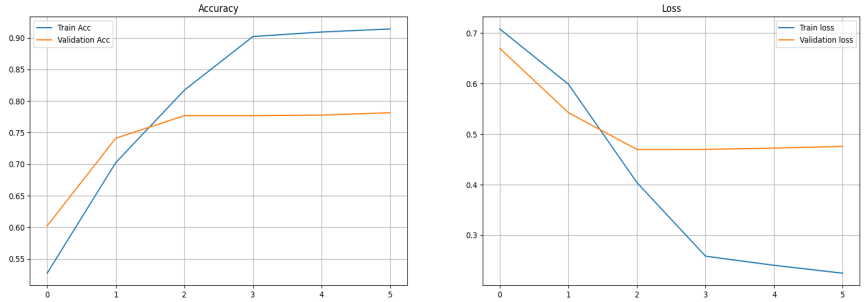


Figure 12

BiGRU/BiLSTM with Self-Attention

Following the success of the BiGRU model and since both types of networks are commonly used and interchanged in research, we have also tried using the same model configuration but adapted for LSTM. Our results show that the LSTM model does perform better with a more complex configuration. There was also an attempt at adding a self-attention layer to reduce the impact of overfitting by replacing the concatenation of the last hidden state in GRU config with the attention output.

Part 3(c) Answers

Test Accuracy scores of BiLSTM and BiGRU model:

	BiLSTM (naïve)	BiLSTM (GRU Config)	BiLSTM with Attention	BiGru	BiGRU with Attention
Test Accuracy	0.771	0.8039	0.8068	0.8011	0.7899

Table 7

Comparing BiLSTM and BiGRU

Overall, due to the small dataset size,it is hard to say which model is the better solution as both results came out to be quite similar. As we increase the dataset size, it can be argued that LSTM would perform marginally better as the added complexity of LSTM might better capture long sequential information compared to GRU (3 gates versus 2 gates) at the cost of performance and training efficiency.

Part 3(d) Discussion on CNN

Convolutional Neural Networks (CNN) is a type of feed-forward neural network that learns features through kernel optimization. Although it has been mainly applied in multidimensional data like images, there have been successful attempts to apply CNN on one-dimensional data like text.

Part 3(d) Single Channel CNN

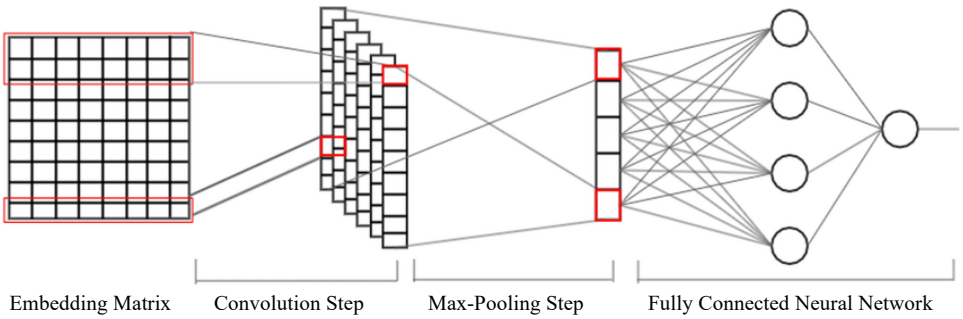


Figure 13: Image Source: Florentin Flambeau, Jiechieu & Tsopze, Norbert. (2021). Skills prediction based on multi-label resume classification using CNN with model predictions explanation.

In our CNN model, we let $x_i \epsilon R^{100}$ be the 100-dimensional word vector (From Glove), where i indicates the i-th word in the sentence. A sentence of length n is represented as $x_{1:n}$ where they are all dynamically padded to accommodate different sentence lengths.

Then, to extract features from this sequence, we apply convolution operation using kernel, where $k \epsilon R^{h\,x\,100}$ and h is the window size. This kernel simulates the processing of local N-gram features by sliding over each window of words during the processing. For each window of words $x_{i:i+h-1}$, we compute a feature this formula:

$$c_i = f(kx_{i:i+h-1} + \mathrm{b}). \quad (1)$$

Here, b is a bias term, which is automatically set to True using Pytorch's `nn.Conv1d`.The convolution step captures relevant patterns from different regions of the sentence, generating a feature map.

Next, we apply max-over-time pooling operation over the feature map and take the maximum value $\hat{c} = max\{c\}$. The idea is to capture the most significant feature and hence we can apply a linear function on the most significant feature to do binary classification.

Part 3(d) Dual Channel CNN

With our working CNN in hand, we decided to implement CNN multichannel for our text classification, inspired by CNN multichannel image processing. In our dual-channel CNN, we use two sets of word embeddings. Initially, the word embeddings will be identical, however, we freeze one of the channels and unfreeze another. So, the model will be able to fine-tune one set of the embeddings while keeping another static. Now, each word vector is represented by $x_i \epsilon\, R^{200} = [x_{trainable,i}, x_{frozen,i}]$.

We hope that the dual channel CNN can prevent the architecture from overfitting, making sure that the learned parameters will not change the model drastically.

Part 3(d) Dual Channel CNN + Attention

In this setting, instead of using max-over-pooling after the convolution step, we replace it with an attention mechanism. After obtaining feature maps from the convolutional layers, we apply an attention layer to calculate importance weights across the features. The final representation of aggregated features, e is computed as a weighted sum over these feature maps

$$e = \Sigma(softmax(e_i) * h_i) \tag{2}$$

where h_i are the features extracted from the i-th word by all convolutional filters and $e_i = W^T h_i + b$ are the attention scores computed from a linear layer.

Part 3(d) Answers

	CNN (Single Channel)	CNN (Dual Channel)	CNN (Dual Channel + Attention)
Test Accuracy	0.7805	0.7917	0.8143

Table 8

Training graph for CNN best result:

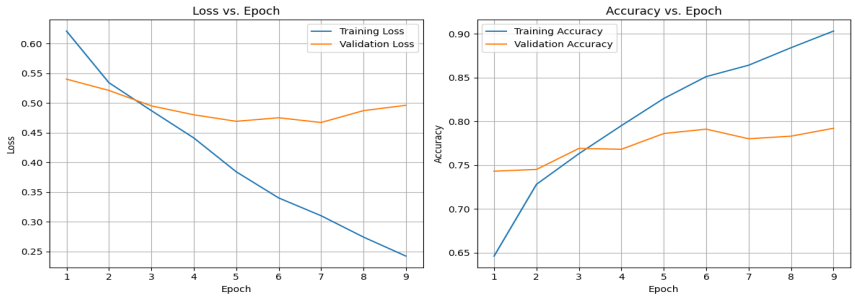


Figure 14

Part 3(e) Discussion

We tried to apply attention to our RNN from Part 2, to enable the model to attend to important words that would affect the overall sentiment. Max pooling or average pooling could be applied to the output, but we found that the performance variance is small for either case. We tried a concatenation of max and average pooling of the hidden state at each time step before feeding it to the FC layer.

We then moved on to build self-attention (SA) networks using the definition from lectures. It was interesting to see that for SA networks, the use of more layers on a small dataset caused performance degradation, and a significant drop in test accuracy can be seen from the 3rd layer onwards. This is despite the use of residual connections to smooth the loss landscape.

We tried adding positional encoding to the self-attention network to capture the index of the words and the average performance gain is marginal if not none. This is interesting as it implies that for this specific dataset, the way the words are ordered schematically is not necessarily important to classifying the sentiment.

Finally, we attempted to apply negation handling in the dataset as an augmentation attempt to expose the model to greater varieties of syntax form. An example would be “I don’t like the movie” to “I dislike the movie”. With the augmentation and application of it to our best model, the performance gain is also not significant, suggesting that the test data did not have such characteristics.

Part 3(e) Answers

Overall, through attempts to improve the model, we found that increasing the complexity of the model caused greater performance drop rather than an improvement. We still found that throughout the experiment the CNN model with Dual channel and attention performed the best. We will, however, still report the best findings from part 3(e) which is our RNN model with attention

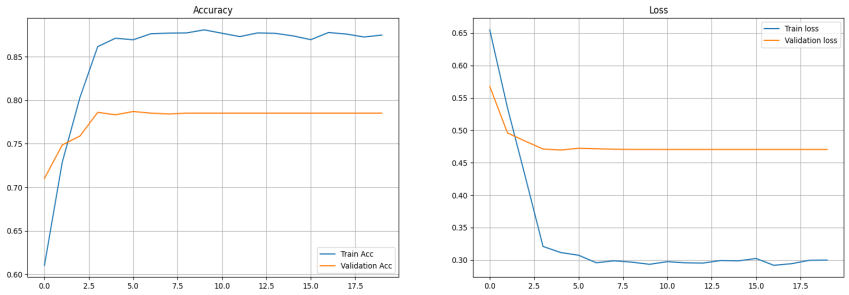


Figure 15

Parameters:
num_layer=1
dropout = 0.5
hidden_dim=100
Max pooling with attention

Test accuracy: 0.814

Part 3(f) Answers

We detail the results in Table 9 and with the improvement in 3e, we observed that overall the RNN with some tweaks performed extremely well, this is from the fact that the dataset is small and tricks such as dropout and attention will help the model attend to the input better.

We would like to comment on the distribution of the dataset as well, the average token length as about 20, with the longest reaching to 60. We reason that the RNN can achieve such a great performance is likely from the fact that vanishing gradient is not an issue when the token length is not sufficiently long. That is also possibly why the LSTM and GRU model only had slight incremental performance compared to the RNN.

It is interesting that of the models, CNN with attention performed the best. CNN alone does not capture the contextual representation, however with an attention mechanism, it adequately provides better loss propagation allowing the model to better learn generalised patterns.

Model	Parameters	Results
Part 2		
RNN -Max pooling on hidden layer	Initial learning rate=0.0001 Batch size=32 Optimizer=Adam Number of training epochs= 19	Test Loss: 0.547 Test accuracy: 0.745 Validation accuracy: 0.703
Optimal RNN - Max pooling on output layer	Initial learning rate=0.001 Batch size=32 Optimizer=Adam Number of training epochs=14 Dropout=0.3 Hidden Size=128 Number of RNN layers=2	Test Loss: 0.490 Test accuracy: 0.773 Validation accuracy: 0.754
Part 3		
RNN - No unknown handling - Unfreeze embedding	Same as Optimal RNN params	Test Loss: 0.509 Test Accuracy: 0.775
RNN - Unknown handling	Same as Optimal RNN params	Test Loss: 0.595 Test Accuracy: 0.783
BiLSTM (Naive)	max_gradclip_norm=5	Test Loss: 0.516

	num_layer =2 dropout=0.3 learning rate=0.0001 hidden_dim=8	Test Accuracy:0.771
BiLSTM (GRU config) - Pooling: Max Pooling - Average Pooling on Hidden Layer - Spatial Dropout after embedding layer	hidden_dim = 32 num_layer = 1 dropout = 0.5 learning_rate = 0.001	Test Loss: 0.4865 Test Accuracy: 0.8039
BiGRU - Pooling: Max Pooling - Average Pooling on Hidden Layer - Spatial Dropout after embedding layer	hidden_dim = 8 num_layer = 1 dropout = 0.5 learning_rate = 0.001	Test Loss: 0.4315 Test Accuracy: 0.8011
CNN - Single Channels - Max pooling	num_filters = 100 filter_sizes = [2, 3, 4, 5, 6, 7] dropout = 0.5 learning_rate = 0.001	Test Loss: 0.4444 Test Accuracy: 0.7805
CNN - Dual Channels - Max pooling	num_filters = 100 filter_sizes = [2, 3, 4, 5, 6, 7] dropout = 0.5 learning_rate = 0.001	Test Loss: 0.4437 Test Accuracy: 0.7917
CNN - Dual Channels - Attention	num_filters = 100 filter_sizes = [3, 5, 7] dropout = 0.5 learning_rate = 0.001	Test Loss: 0.4172 Test Accuracy: 0.8143
RNN - Max pooling - Attention	num_layer=1 dropout = 0.5 hidden_dim=100	Test Loss: 0.4190 Test Accuracy: 0.8142
RNN - Max pooling - Ave pooling - Concatenate both results	num_layer=1 dropout = 0.5 hidden_dim=100	Test Loss: 0.6042 Test Accuracy: 0.7821
Self attention	num_layer=1 dropout = 0.5 hidden_dim=100	Test Loss: 1.487 Test Accuracy: 0.760
Self attention - Positional encoding	num_layer=1 dropout = 0.5 hidden_dim=100	Test Loss: 0.524 Test Accuracy: 0.750

Table 9

Another interesting finding is that average pooling on the dataset performs worse off than max pooling, suggesting that classifying the sentiment of the sentence can be done through a significant word that carries a strong connotation for the case of the RNN. This technique when applied to the attention network, however, does not confer the same advantage, instead a mean of the output produced a more meaningful result.

We are of the opinion that the dataset is relatively small and the use of more complex models such as LSTM and GRU is unable to capture the schematics of the corpus, especially when there are reviews which are written informally. What we think might help would be helpful could be to explore the following

1. Dataset augmentation to increase corpus size which reduces tendency of overfitting
2. Apply a pre-trained transformer model and fine tune the parameters on the train dataset. This generalised model would likely to perform much better than the pre-trained word embeddings as we are transferring the words-in-context