NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC4002 Natural Language Processing

Project Report

Contributor
Kakuly Mittal
Gupta Tushar Sandeep
Abhishekh Pandey
Anapana Dinesh Kumar

Contents

Objectives
The aim of this report is to showcase the methodology, experiments, results and evaluation of the iterative process taken to build a general classification system on top of the existing pre-trained word embeddings (e.g., word2vec and Glove). The system includes features such as sentence (sentiment) classification tasks. This report outlines our use of typical NLP applications such as word embeddings and typical deep learning models for sentence classification tasks along with the iterative process of training and evaluating effective classifiers from pre-trained word embeddings.

Background
For more information on Glove, Word2Vec, FastText and Lemmatization refer to the Appendix.

Advantages of using FastText over Glove and Word2Vec: This makes it considerably more effective than Glove or Word2Vec since these cannot provide embeddings for words that do not appear while training data and in the model's vocabulary.

About Lemmatization: It is a text pre-processing technique which is used in natural language processing to break a word down to its root meaning in order to identify similarities.

Part 1: Preparing Word Embeddings

Answer 1. a)

- The size of the vocabulary formed during our training data using Word2Vec is **16545**.
- The size of the vocabulary formed during our training data using Glove is **18950**.

Answer 1. b)

- The number of OOV words that exist in our training data using Word2Vec is **1651**.
- The number of OOV words that exist in our training data using Glove is **3034.**

Methodology
First, we loaded the pre-trained embeddings for both GloVe and Word2Vec. The GloVe model, specifically the 100-dimensional version, is loaded using the torchtext.vocab.GloVe library, which provides efficient access to GloVe embeddings. In parallel, we load the 300-dimensional Word2Vec model from Gensim's pre-trained library, known for capturing richer word relationships through its higher dimensionality. Next, we process the training dataset to build a vocabulary. We tokenize each text example using WordPunctTokenizer, splitting the text into individual words. Each word is then converted to lowercase to ensure consistency with the pre-trained GloVe and Word2Vec vocabularies, which often store words in lowercase format. This normalized word is added to a vocab set, enabling us to accumulate a comprehensive list of unique words in the training data. To manage embeddings for these words, two embedding functions, GloVe_embedder and Word2Vec_Embedder, are defined. The GloVe_embedder function checks each word against the GloVe vocabulary (glove.stoi). If the word exists in GloVe, its vector is added to word_embeddings; if not, the word is recorded in an OOV set (oov_words_GloVe) and assigned a random 300-dimensional vector as a placeholder. Similarly, the Word2Vec_Embedder function verifies the presence of each word in the Word2Vec model. Words found in Word2Vec are stored in word_embeddings, while absent words are added to oov_words_Word2Vec and assigned a random 300-dimensional vector.

Once the embeddings are created, they are saved as .pth files (word_embeddings_GloVe.pth and word_embedding_word2vec.pth) using torch.save. Finally, we calculate and print the vocabulary size and the number of OOV words for both models and export the lists of OOV words to Excel files for any potential further analysis. In the GloVe code, we first load the GloVe embeddings and set up necessary data structures, including word_embeddings, vocab, and oov_words_GloVe. We iterate through the training data, tokenize each sentence, and lowercase each word. For each word, the GloVe_embedder function is called to check its presence in GloVe. Found words are embedded while missing words are stored as OOV. The final embeddings and OOV words are saved and exported. The Word2Vec code follows a similar pattern: the Word2Vec model is loaded, and text tokenization and processing occur for each word in the dataset. The Word2Vec_Embedder function adds embeddings for known words and stores random vectors for OOV words. The embeddings and OOV words are then saved and exported, completing the Word2Vec processing.

Answer 1.c) Best strategy to mitigate such limitation

Corresponding code snippets have been added below. Please refer to **Figures 1-10 in Appendix.** FastText is built upon Word2Vec's architecture, making their embeddings naturally compatible while Word2Vec treats the words as atomic units, FastText extends this by breaking words into character n-grams, allowing for better handling of morphologically rich languages. Word2Vec excels at capturing semantic relationships between complete words and their contexts. FastText's subword information allows it to generate embeddings even for words not seen during training by utilizing character n-gram patterns. Both of these together provide both word-level and subword-level representations.

Tokenization is a crucial step in natural language processing as it helps to ensure that the text is split into words that can be mapped to embeddings. With the help of Word2PunctTokenizer, the tokenizer ensures that punctuation is treated as separated tokens and words are split accordingly. Since word embeddings are case-sensitive, we convert each word to lowercase so as to standardize it, thus ensuring consistency. We first find the embeddings for each word from the Word2Vec model and FastText. If a word exists in one of these models, we will assign the respective embedding. Moreover, if the words are not found in the embeddings, we implement lemmatization. Lemmatization is used to reduce the word to its base form. The presence of this lemmatized word is then checked in both the Word2Vec and FastText models. This approach helps us to address those words that may not appear in their surface form but can be found in their base form. Also, the digits are handled separately and are assigned a random embedding to ensure that they are represented correctly in the word embeddings. We remove those words which are composed of non-alphanumeric characters/symbols since they do not provide meaningful information. *Note:* The reason why we put remove symbols before edit distance was so that we can remove "words" that contain only 2 symbols, eg. "%@" before we calculate edit distance from them because it could give us a false positive for any 1 or 2 character words. We prioritized this placement of symbol removal for computational efficiency, as it prevents the need to iterate through each character for every word, thereby reducing unnecessary processing overhead. We also implemented another strategy, which is using Levenshtein edit distance to find the closest matching word in either Word2Vec or FastText. If the word is not found using the above-mentioned strategies, the algorithm computes the closest word from the model's vocabulary by checking the edit distance (with a threshold of 2) to determine similarity.

Part 2. Model Training & Evaluation - RNN

Q2a) Best Results Across all Depths

Depth	Width	Model Performance	Model Details
1	256	Test loss: 0.5293 Test Accuracy: 74.6% Early Stopping after 20 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
2	64	Test loss: 0.4967 Test Accuracy: 76.1% Early Stopping after 30 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
3	64	Test loss: 0.4831 Test Accuracy: 77.1% Early Stopping after 29 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64

Evaluation

The model that performed the best was the RNN model with attention mechanism used. Again, word2vec embeddings were selected, with the embeddings unfrozen. It seems that with a depth of 3 layers, the model is able to capture sufficiently complex patterns in the data, whereas if the layers are too deep, it might cause overfitting. The width of 64 units is a good balance between capacity and generalising, focusing on the important parts of the input sequence. The Test Accuracy is 77.1% while the Test Loss is 0.4831. Early stopping occurred after 29 epochs. The initial learning rate set was 0.0001 using Adam optimiser. The batch size used was 64.

Q2b) Best Configuration Results

The configuration of the best RNN model was the training model with depth=3 and width=64. Early stopping was triggered after 29 epochs. The test loss is 0.4831 and the test Accuracy is 77.1%. The accuracy score on the test and validation set for each epoch during training is reflected in the table below. We evaluated that the convention is for the model to see the test set only once as the weights are only updated based on the training and validation set. Hence we ran the model only once on the test set after training was done.

```
Epoch 1/100 - Training loss: 0.6934, Validation loss: 0.6926
Training Accuracy: 49.6%, Validation Accuracy: 50.5%
Epoch 2/100 - Training loss: 0.6922, Validation loss: 0.6928
Training Accuracy: 51.9%, Validation Accuracy: 50.5%
Epoch 3/100 - Training loss: 0.6914, Validation loss: 0.6921
Training Accuracy: 51.7%, Validation Accuracy: 51.8%
Epoch 4/100 - Training loss: 0.6889, Validation loss: 0.6877
Training Accuracy: 52.7%, Validation Accuracy: 52.5%
Epoch 5/100 - Training loss: 0.6435, Validation loss: 0.6254
Training Accuracy: 63.6%, Validation Accuracy: 65.9%
Epoch 6/100 - Training loss: 0.5840, Validation loss: 0.5790
Training Accuracy: 70.3%, Validation Accuracy: 69.4%
Epoch 7/100 - Training loss: 0.5528, Validation loss: 0.5582
Training Accuracy: 72.5%, Validation Accuracy: 71.1%
Epoch 8/100 - Training loss: 0.5417, Validation loss: 0.5873
Training Accuracy: 73.0%, Validation Accuracy: 69.8%
Epoch 9/100 - Training loss: 0.5195, Validation loss: 0.5382
Training Accuracy: 74.5%, Validation Accuracy: 71.7%
Epoch 10/100 - Training loss: 0.5080, Validation loss: 0.5387
```

```
Epoch 16/100 - Training loss: 0.4763, Validation loss: 0.5214
Training Accuracy: 77.1%, Validation Accuracy: 73.0%
Epoch 17/100 - Training loss: 0.4709, Validation loss: 0.5213
Training Accuracy: 76.9%, Validation Accuracy: 73.5%
Epoch 18/100 - Training loss: 0.4783, Validation loss: 0.5266
Training Accuracy: 76.7%, Validation Accuracy: 74.1%
Epoch 19/100 - Training loss: 0.4657, Validation loss: 0.5448
Training Accuracy: 77.6%, Validation Accuracy: 73.5%
Epoch 20/100 - Training loss: 0.4646, Validation loss: 0.5188
Training Accuracy: 77.4%, Validation Accuracy: 73.5%
Epoch 21/100 - Training loss: 0.4604, Validation loss: 0.5933
Training Accuracy: 77.8%, Validation Accuracy: 72.5%
Epoch 22/100 - Training loss: 0.4609, Validation loss: 0.5155
Training Accuracy: 77.6%, Validation Accuracy: 72.9%
Epoch 23/100 - Training loss: 0.4554, Validation loss: 0.5195
Training Accuracy: 78.0%, Validation Accuracy: 72.9%
Epoch 24/100 - Training loss: 0.4529, Validation loss: 0.5151
Training Accuracy: 78.4%, Validation Accuracy: 73.1%
Epoch 25/100 - Training loss: 0.4545, Validation loss: 0.5154
```

```
Training Accuracy: 75.3%, Validation Accuracy: 71.8%
Epoch 11/100 - Training loss: 0.4945, Validation loss: 0.5358
Training Accuracy: 76.2%, Validation Accuracy: 73.0%
Epoch 12/100 - Training loss: 0.4913, Validation loss: 0.5270
Training Accuracy: 76.1%, Validation Accuracy: 71.6%
Epoch 13/100 - Training loss: 0.4908, Validation loss: 0.5372
Training Accuracy: 75.9%, Validation Accuracy: 73.2%
Epoch 14/100 - Training loss: 0.4817, Validation loss: 0.5233
Training Accuracy: 76.9%, Validation Accuracy: 72.5%
Epoch 15/100 - Training loss: 0.4759, Validation loss: 0.5206
Training Accuracy: 77.0%, Validation Accuracy: 72.9%
```

```
Training Accuracy: 78.1%, Validation Accuracy: 73.5%
Epoch 26/100 - Training loss: 0.4509, Validation loss: 0.5198
Training Accuracy: 78.3%, Validation Accuracy: 73.5%
Epoch 27/100 - Training loss: 0.4543, Validation loss: 0.5242
Training Accuracy: 78.1%, Validation Accuracy: 73.7%
Epoch 28/100 - Training loss: 0.4463, Validation loss: 0.5166
Training Accuracy: 78.3%, Validation Accuracy: 74.0%
Epoch 29/100 - Training loss: 0.4465, Validation loss: 0.5261
Training Accuracy: 78.8%, Validation Accuracy: 73.3%
Early Stopping Triggered
Test loss: 0.4831, Test Accuracy: 77.1%
```

Q2c) Results for Different Model Architectures
Mean-pooling was done by calculating the mean of all the word embeddings within each sentence so as to establish a single-sentence representation. The average word representation is assumed to be able to sufficiently capture the overall sentiment context of the sentence. This helps us handle sentences of varying lengths as they are standardised into a vector of a fixed size. The **accuracy score on the test set was 79.2%** with a **test loss of 0.4613.**

Max-pooling was done by taking the maximum value in each dimension of the word embeddings within a sentence such as to identify the most significant features within the sentence. This does so by looking at the extreme values of each dimension. A potential drawback of this method is that by focusing on the extremes, subtle nuances within the sentence that hold contextual information might be overlooked. This might explain why the **accuracy score on the test set was 76.8%**, slightly lower than that of mean-pooling and a slightly higher **test loss of 0.4905.**

In order to add a layer of complexity, an attention mechanism was added to the RNN. Attention helps the model to learn the weight of every word's contribution to the sentence representation in a dynamic manner. It assigns attention weights that place higher value on words that bear more sentiment while placing less value on words with a neutral nuance. The results from this strategy were the best, with the highest **accuracy score on the test set of 79.3%** and the lowest **test loss of 0.4585.**

Different strategies are used because the simple RNN uses a hidden state of the last word in the sequence as the sentence representation. While this is an efficient method, it may not effectively capture long-term dependencies as well, especially if the sentiment-bearing words are far away from the end of the sentence. The **accuracy score on the test set was 78.1% with a test loss of 0.6928.**

It is essential to note that for all the models used while performing preprocessing, a max sentence length of 30 was used. This was done to standardise the lengths of the sentences. Upon checking the distribution of the sentence lengths in the training set, the 50th percentile was 23 words and the 90th percentile was 34 words. We chose to keep a maximum sentence length of 30 in order to retain key information without the unnecessary elements of the sentence. This was a good trade-off between retention of useful information and computational efficiency. In later parts of the assignment, mean pooling, max pooling, and the addition of an attention mechanism were also applied to the other models, like biLSTM, biGRU, etc.

Part 3: Enhancement

Q3a) Best Results across all Aggregation Methods

Experiment	Model Performance	Model Details
rnn_model_tuning_updated_embeddings	Test Accuracy: 78.1% Test Loss: 0.6928 Early stopping after 7 epochs	Depth: 1 Width: 256 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Model_attention_tuning_updated_embeddings	Test Accuracy: 76.7% Test Loss: 0.6171 Early Stopping after 7 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Mean_pooling_tuning_updated_embeddings	Test Accuracy: 76.8% Test Loss: 0.5536 Early Stopping after 10 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Max_pooling_tuning_updated_embeddings	Test Accuracy: 77.1% Test Loss: 0.5163 Early Stopping after 12 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64

In these models, the word2vec embeddings were unfrozen and the model was run again. After unfreezing the embedding (for part 3.1), the model needs fewer epochs to train and reach convergence. One reason is that they now act as parameters that can be trained and fine-tuned. The model is able to understand specific representations and arrive at an optimal state in lesser epochs. Accuracy remains the same when it is unfrozen as the pre-trained embeddings enable sufficient semantic information from the original corpus. While the accuracy change is insignificant, the training speed is much more efficient.

Q3b) Accuracy score on test set with our method to deal with OOV words in Part 3.2.

Experiment	Model Performance	Model Details
rnn_model_tuning_updated_embeddings	Test Accuracy: 79.1% Test Loss: 0.5894 Early stopping after 12 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Model_attention_tuning_updated_embeddings	Test Accuracy: 77.5% Test Loss: 0.6115 Early Stopping after 10 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Mean_pooling_tuning_updated_embeddings	Test Accuracy: 76.5% Test Loss: 0.6099 Early Stopping after 10 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64
Max_pooling_tuning_updated_embeddings	Test Accuracy: 77.0% Test Loss: 0.5107 Early Stopping after 9 epochs	Depth: 1 Width: 64 Max Epochs: 100 Learning Rate: Default Optimizer: Adam Batch Size: 64

Q3c) BiLSTM

Introduction

BiLSTM model is an advanced type of RNN that processes data in both forward and backward directions to capture context from both past and future states. It consists of two LSTM layers that process the input sequence in opposite directions. One LSTM layer processes the sequence from start to end (forward direction), while the other processes it from end to start (backward direction).

Our Results:

We expected the test accuracy to increase using BiLSTM. This is because BiLSTMs are specifically designed to capture context from both past and future states in a sequence, whereas a traditional RNN only captures information from past states. BiLSTM also addresses the vanishing gradient problem better than vanilla RNNs. In traditional RNNs, gradients can diminish over long sequences, causing the model to struggle with learning dependencies over extended text. The LSTM structure, with its gated mechanisms, helps to maintain these dependencies, making it more suitable for sentiment analysis, which often requires understanding relationships across an entire sentence or even multiple sentences.

However, in practice, we noticed that the improvement in accuracy was not significant. This could be due to overfitting, especially since our dataset is relatively small. BiLSTMs, with their additional parameters, require more data to generalize well.

Aggregation Method	Batch Size	Depth	Width	Validation Accuracy (%)	Test Accuracy (%)	RNN Model without BiLSTM
Normal	64	2	256	76.5%	79.3%	79.1%
Attention	128	1	256	76.8%	79.9%	77.5%
Max Pooling	128	1	128	76.3%	80.9%	77.0%
Mean Pooling	32	1	64	77%	80.0%	76.5%

Table: 3.b.i) Best Test Accuracy Results for biLSTM

Bi-GRU

Our Results:

Aggregation Method	Batch Size	Depth	Width	Test Accuracy (%)	RNN Model
Normal	64	1	64	78.0%	79.1%
Attention	64	2	64	80.4%	77.5%
Max Pooling	64	1	64	80.9%	77.0%
Mean Pooling	32	1	64	80.4%	76.5%

Table: 3.c.i) Best Test Accuracy Results for BiGRU

Result Analysis:
The importance of further tuning is clearly demonstrated by the fact that max, mean and attention pooling models scored a higher accuracy on average than the normal model. It appears that max pooling has gained the highest accuracy. This may be due to the model's ability to capture the most informative features across the sequence. Max pooling highlights the strongest activation in each feature dimension, allowing the model to focus on the most prominent signals, which often correspond to the critical sentiment-carrying words. This enhances sentiment detection by preserving the most impactful features in the sequence, effectively capturing the essence of sentiment without diluting it with less significant details. In contrast, mean pooling averages all features, which may smooth out important information, while attention pooling distributes focus across the sequence but may not always prioritize the most sentiment-relevant words as effectively as max pooling. This difference in capturing key information likely contributed to the superior performance of the biGRU Max pooling model.

Nevertheless, all of them give quite comparable results, which indicates that, if one wants to gain even higher accuracy levels, more advanced pooling and tuning methods may need to be incorporated.

Relevance & Comparison of BiLSTM and Bi-GRUs to Sentiment Analysis:
Upon comparing BiLSTM and BiGRU, BiGRUs often work better across all the aggregation methods. In our case, this can possibly be associated with the fact that biGRU performs better on smaller datasets because of its simpler architecture. It does not risk overfitting as much as an LSTM might.

BiLSTMs and Bi-GRUs are highly relevant to sentiment analysis due to their ability to capture context from both past and future states in a sequence. This bidirectional processing is crucial for understanding the sentiment expressed in text, as the meaning of words can depend heavily on their surrounding context. For instance, in a sentence like "I don't think this movie is bad," the word "bad" is negated by the preceding words, and a unidirectional model might miss this nuance. By processing the text in both directions, biGRUs and biLSTMs can better understand such dependencies, leading to more accurate sentiment predictions (Dhingra & Cheung, 2021).

Q3d) Report the accuracy score of CNN on the test set.
The accuracy score received on applying Convolutional Neural Network on the test set is *78.6%* for depth 1, width 64 and batch size 64.

Results across different combinations of widths are shown in the below table.
a) output_dim = 1 b) num_filters = 100 c) kernel_sizes = [3, 4, 5]

Batch Size	Depth	Width	Test Accuracy (%)
64	1	64	78.6%
64	1	128	77.9%
64	1	256	76.8%

Table: 3.d.i) Test Accuracy Results for lr = 0.0001 and different batch sizes

In our CNN model, it uses pre-trained word embeddings from the FastText model. And the embeddings are set to be trainable during the training process. The words that are not found in the vocabulary are mapped to a special <UNK> token, and the text data is then padded or truncated to a fixed length of max_length=30 to ensure all samples have the same dimensions. A dictionary vocab_cnn is created to map each unique word to a unique integer index. This allows the text data to be represented as a sequence of integer indices. The CNN model uses three parallel convolutional layers with different kernel sizes (3, 4, and 5). This allows the network to learn different types of features from the word embeddings, such as bigrams (2-grams), trigrams (3-grams) and four-grams. For each kernel size, a separate convolutional filter is created using the nn.Conv2d layer. This layer has one input channel (since the input is a 1D sequence of word embeddings) and a user-defined number of filters (num_filters), which are set to 100 by default. The output of the convolutional layers is then passed through a ReLU activation function to introduce non-linearity. This is to ensure that only positive activations are passed to the next layer. We then apply max pooling to each feature map, selecting the maximum value within each filter. This extracts the most important features from the convolutional outputs and reduces the dimensionality of the features. To prevent overfitting, we apply a dropout layer = 0.5. Dropout randomly drops neurons during training and makes the model more robust. BCEWithLogitsLoss is implemented which combines the sigmoid activation and binary cross-entropy loss in a single module. This is used for providing stability in sentiment score prediction. The output of the model is a single value representing the sentiment score for the input text.

We have found that Bi-GRU and Bi-LSTM are achieving higher accuracy than CNN models. Bidirectional GRU (BI-GRU) and Bidirectional LSTM (BI-LSTM) often achieve higher accuracy than Convolutional Neural Networks (CNNs) in sequence-based tasks due to their ability to capture temporal dependencies in both forward and backward directions. While CNNs excel at extracting spatial features and are highly effective for image processing tasks, they are less suited for handling sequential data where the order of elements is crucial. BI-GRU and BI-LSTM, on the other hand, are designed to process sequences by maintaining information from both past and future contexts, which is particularly beneficial for tasks like natural language processing, speech recognition, and time-series prediction. This bidirectional approach allows them to understand the context more comprehensively, leading to more accurate predictions and better performance in tasks that require understanding the sequence's structure and dependencies.

Answer 3.e) Final improvement strategy and model performance.

We have implemented the Bidirectional Encoder Representations from Transformers (BERT) model as our final improvement strategy.

BERT is an open-source deep learning model introduced by researchers at Google in October 2018. It is used for representing text as a sequence of vectors using self-supervised learning. It is a novel pre-training approach that allows the model to learn contextual word representations by considering both the left and right context simultaneously.

In our BERT model, we have implemented and leveraged the BertForSequenceClassification module from the Hugging Face Transformers library, specifically the bert-base-uncased checkpoint. This module is fine-tuned to perform binary classification, using the pre-trained BERT weights which are then adapted to classify input text into one of two classes. We have performed tokenization using the BertTokenizer from the Transformers library which is used to convert text into tokens. Furthermore, padding and truncation are applied to maintain a uniform sequence length, set at a maximum length of 30 tokens per sentence. BERT uses an embedding matrix to convert the tokenized text into dense vectors. After the token is transformed through subword segmentation, the token is mapped to an embedding vector from this matrix. BERT uses three types of word embeddings that are combined: i) Token embeddings for representing the meaning of each token. ii) Segment embeddings to distinguish between pairs of sentences. iii) Position embeddings for capturing the position of each token in the sequence. All of the above embeddings are fed as input to the BERT's encoder layers, which allow the model to capture the contextual representation of each token. BERT uses transformer layers instead of convolutional layers. This transformer layer consists of a multi-head self-attention mechanism, feed-forward neural networks, layer normalization and residual connections. The primary activation function used in the BERT model is GELU (Gaussian Error Linear Unit). This is applied after the linear transformations within each layer. GELU is chosen for its ability to perform well with the Transformer architectures as it provides a smooth, non-linear transformation that is effective for large-scale neural networks. We have applied attention pooling and the dropout method to prevent overfitting. Dropout randomly masks portions of the network, which helps in model generalization, especially when fine-tuning BERT on domain-specific data. The final layer is a fully connected layer that maps the hidden states to the number of classes In our implementation. We have used 2 output neurons for binary classification. The [CLS] token's embedding at the final layer is passed through this fully connected layer because it captures the contextualized information of the entire sentence. We have applied the Cross-Entropy Loss function as it helps in measuring the divergence between predicted labels and actual labels. During training, BERT minimizes the loss and adjusts weights in order to increase its classification. The model is trained using the Hugging Face Trainer class, which handles: batch processing, learning rate scheduling, gradient updates and model evaluation. Performance metrics such as F1-score and accuracy are used to evaluate model performance. Since our dataset is quite small for the architecture of BERT, the model starts overfitting within epochs. Hence we have implemented early stopping with a patience of 2 epochs to prevent overfitting.

The accuracy of using our improved model is:

No.	Batch Size	Test Accuracy (%)
1	32	83.1%
2	64	84.2%

We observe that BERT has an improved performance of 84.2%, which is not that significant in comparison to our results from BiGRU and BiLSTM. A major reason is that our dataset is quite small and not that complicated. Whereas BERT's complicated architecture is known for its ability to perform well on large datasets and because BiGRU/BiLSTM can capture simpler patterns effectively without needing the complex architecture of BERT. Because of the smaller dataset, BERT also tends to overfit within the first two epochs of the model. For future iterations, we suggest a lighter BERT architecture like DistilBERT (97% of BERT's performance while using 40% fewer parameters) can be used as it would be more appropriate for our dataset size and could help mitigate overfitting while reducing computational requirements. On the other hand, data augmentation techniques could be applied to improve the complexity of our dataset. This dataset with increased complexity can then be utilized to train DistilBERT to take advantage of its more complicated architecture.

Answer 3f) Compare the results across different solutions.

Given that our dataset is relatively small, one important consideration for improving RNN model performance is implementing a drop-out function. Dropout is a regularization technique that helps prevent overfitting by randomly disabling a fraction of the neurons during training, forcing the model to learn more robust features. In our current approach, dropout was implemented in the BiLSTM, BiGRU and within the Stacked RNN and CNN modes, but not in the simple RNN models. Given the small size of the dataset, adding dropout to the BiLSTM architecture may help improve its generalization by preventing it from fitting too closely to the noise in the training data. This is particularly crucial as the model could easily memorize the limited examples in the dataset rather than learn meaningful patterns.

Another aspect to consider is that simpler architectures might perform better with smaller datasets. Our experiments with more complex models, such as BERT, showed that these changes did not result in a significant improvement in performance. In fact, the more complex models did not outperform simpler architectures by large margins. This suggests that adding layers and increasing hidden representation sizes could lead to overfitting rather than improving the model's ability to generalize. For smaller datasets, it is often beneficial to keep the model architecture simpler to avoid overfitting and to maintain better generalization.

To further improve the model, data augmentation techniques could also be explored. By artificially expanding the dataset, for example, through text generation, paraphrasing, or introducing slight variations in the text (such as changing word order or replacing synonyms), we could provide the model with more diverse training examples. This would help the model learn more robust representations, potentially leading to improved accuracy while also mitigating the risks of overfitting, especially when working with small datasets.

References:

Dhingra, K., & Cheung, N.-M. (2021). BGT-Net: Bidirectional GRU Transformer Network for Scene Graph Generation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (pp. 1-10).

Maini, P., Kolluru, K., Pruthi, D., & Mausam, N. (2020). Why and when should you pool? Analyzing Pooling in Recurrent Architectures. *ACL Anthology*, 4568–4586. https://doi.org/10.18653/v1/2020.findings-emnlp.410

Onan, A. (2022). Bidirectional convolutional recurrent neural network architecture with group-wise enhancement mechanism for text sentiment classification. Journal of King Saud University - Computer and Information Sciences, 34(5), 2098–2117. https://doi.org/10.1016/j.jksuci.2022.02.025

Papers with Code - BiGRU Explained. (n.d.). https://paperswithcode.com/method/bigru

Appendix:

About Glove:

a) It was developed as an open-source project at Stanford and launched in 2014.
b) It is an unsupervised learning algorithm for obtaining vector representations for words.
c) Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

About Word2Vec:

a) It was developed by researchers in Google in 2013. It is a word embedding technique to represent the words in a continuous vector form.
b) These vectors capture the semantic and syntactic relationships between words.
c) It learns the word embeddings through an iterative training process using a large corpus of text data.

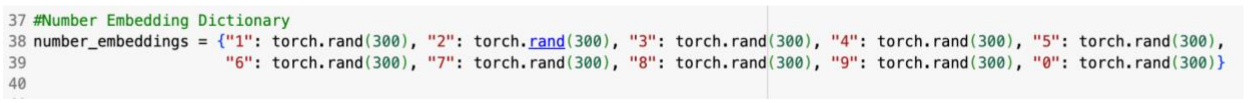
About FastText:

a) FastText was introduced by Facebook’s AI Research (FAIR) team in 2016.
b) It is a library for efficient learning of word representations and sentence classification, which are its two use cases.
c) FastText can represent words that it has never seen before using n-grams. This is quite valuable for models with limited vocabulary.
d) Since FastText provides embeddings for character n-grams, it can also provide embeddings for Out of Vocabulary (OOV) words. If such a word occurs, then FastText provides the necessary embedding for that particular word by embedding its character n-gram.

Depth	Width	Model Performance	Model Details

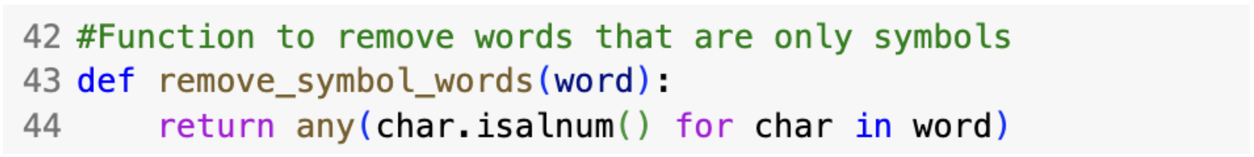
1	64	Test loss: 0.5249 Test Accuracy: 73.5% Early Stopping after 29 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
1	128	Test loss: 0.5287 Test Accuracy: 73.1% Early Stopping after 17 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
1	256	Test loss: 0.5293 Test Accuracy: 74.6% Early Stopping after 20 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
2	64	Test loss: 0.4967 Test Accuracy: 76.1% Early Stopping after 30 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
2	128	Test loss: 0.5219 Test Accuracy: 73.9% Early Stopping after 15 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
2	256	Test loss: 0.5691 Test Accuracy: 70.8% Early Stopping after 14 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
3	64	Test loss: 0.4831 Test Accuracy: 77.1% Early Stopping after 29 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
3	128	Test loss: 0.5145 Test Accuracy: 75.1% Early Stopping after 15 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64
3	256	Test loss: 0.5741 Test Accuracy: 73.4% Early Stopping after 15 epochs	Max Epochs: 100 Learning Rate: 0.001 Optimizer: Adam Batch Size: 64

<u>Table for 2(a) Results</u>



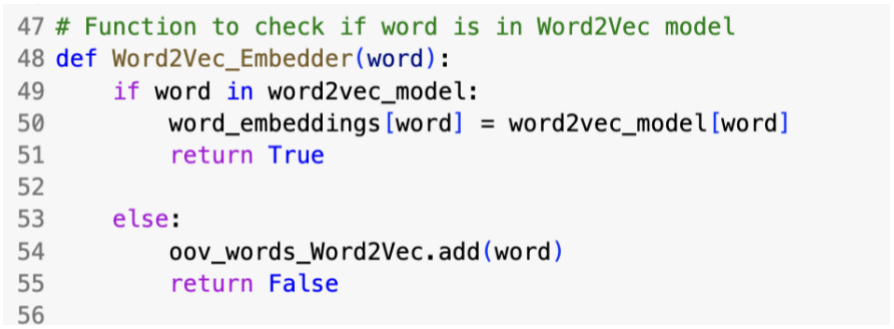
```python
#Number Embedding Dictionary
number_embeddings = {"1": torch.rand(300), "2": torch.rand(300), "3": torch.rand(300), "4": torch.rand(300), "5": torch.rand(300),
                     "6": torch.rand(300), "7": torch.rand(300), "8": torch.rand(300), "9": torch.rand(300), "0": torch.rand(300)}
```

Figure 1: Number Embeddings



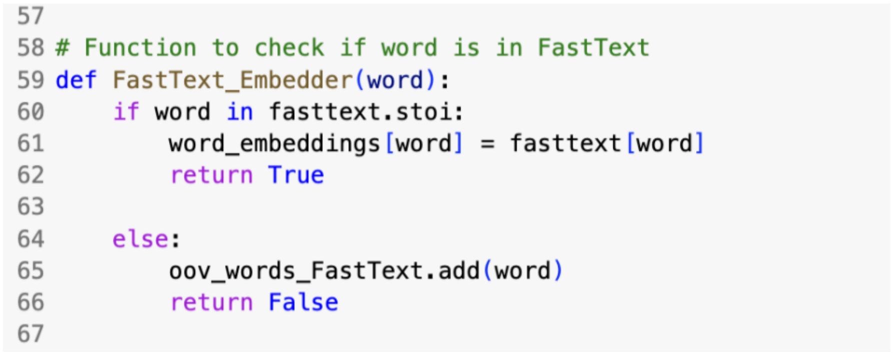
```python
#Function to remove words that are only symbols
def remove_symbol_words(word):
    return any(char.isalnum() for char in word)
```

Figure 2: Removing words that are only symbols



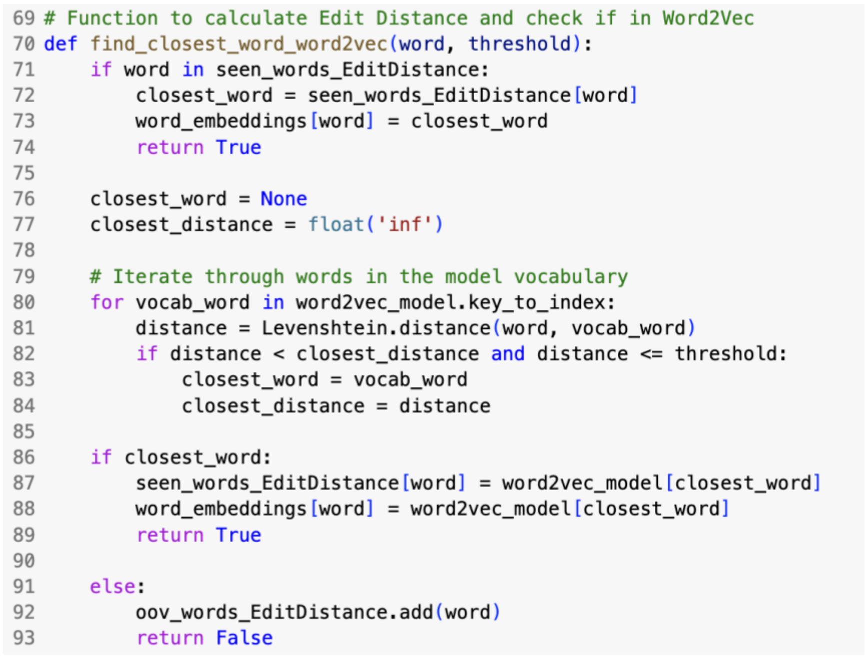
```python
# Function to check if word is in Word2Vec model
def Word2Vec_Embedder(word):
    if word in word2vec_model:
        word_embeddings[word] = word2vec_model[word]
        return True

    else:
        oov_words_Word2Vec.add(word)
        return False

```

Figure 3: Check whether word is in Word2Vec



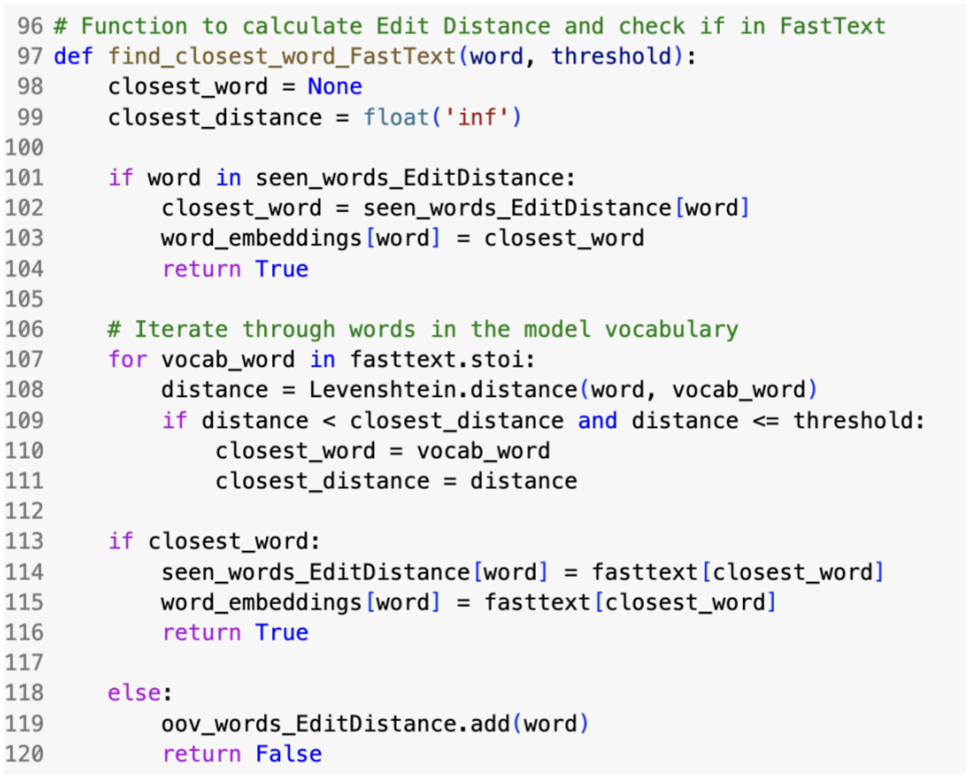
```python

# Function to check if word is in FastText
def FastText_Embedder(word):
    if word in fasttext.stoi:
        word_embeddings[word] = fasttext[word]
        return True

    else:
        oov_words_FastText.add(word)
        return False

```

Figure 4: Check whether word is in FastText



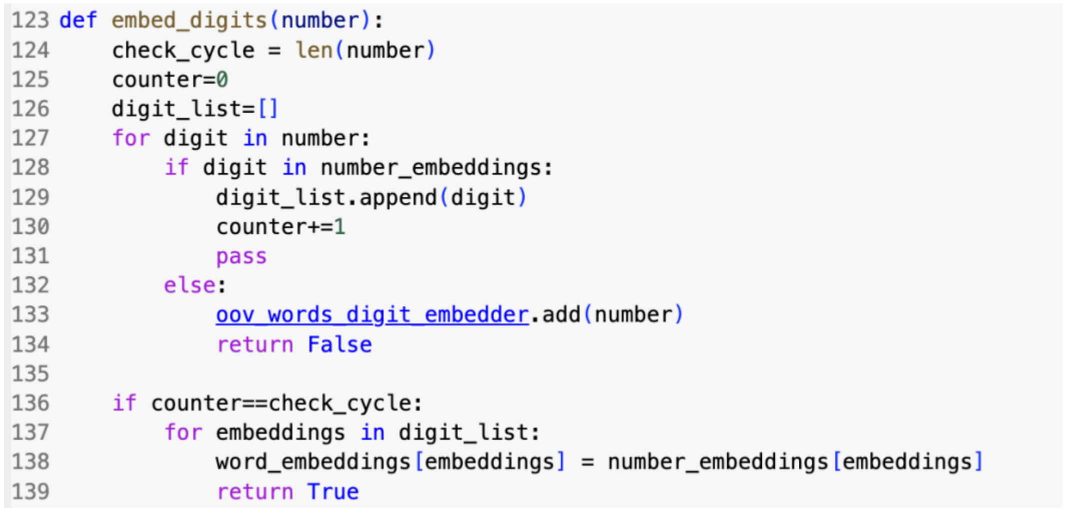
```python
# Function to calculate Edit Distance and check if in Word2Vec
def find_closest_word_word2vec(word, threshold):
    if word in seen_words_EditDistance:
        closest_word = seen_words_EditDistance[word]
        word_embeddings[word] = closest_word
        return True

    closest_word = None
    closest_distance = float('inf')

    # Iterate through words in the model vocabulary
    for vocab_word in word2vec_model.key_to_index:
        distance = Levenshtein.distance(word, vocab_word)
        if distance < closest_distance and distance <= threshold:
            closest_word = vocab_word
            closest_distance = distance

    if closest_word:
        seen_words_EditDistance[word] = word2vec_model[closest_word]
        word_embeddings[word] = word2vec_model[closest_word]
        return True

    else:
        oov_words_EditDistance.add(word)
        return False
```

Figure 5: Calculate Edit Distance and check if it is in Word2Vec



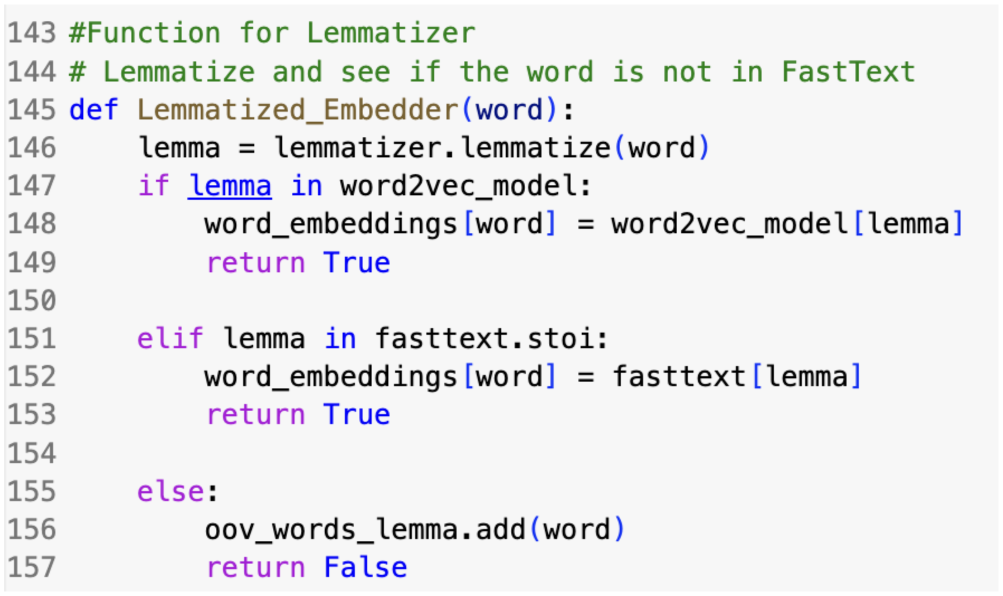
```python
# Function to calculate Edit Distance and check if in FastText
def find_closest_word_FastText(word, threshold):
    closest_word = None
    closest_distance = float('inf')

    if word in seen_words_EditDistance:
        closest_word = seen_words_EditDistance[word]
        word_embeddings[word] = closest_word
        return True

    # Iterate through words in the model vocabulary
    for vocab_word in fasttext.stoi:
        distance = Levenshtein.distance(word, vocab_word)
        if distance < closest_distance and distance <= threshold:
            closest_word = vocab_word
            closest_distance = distance

    if closest_word:
        seen_words_EditDistance[word] = fasttext[closest_word]
        word_embeddings[word] = fasttext[closest_word]
        return True

    else:
        oov_words_EditDistance.add(word)
        return False
```

Figure 6: Check whether word is in Word2Vec



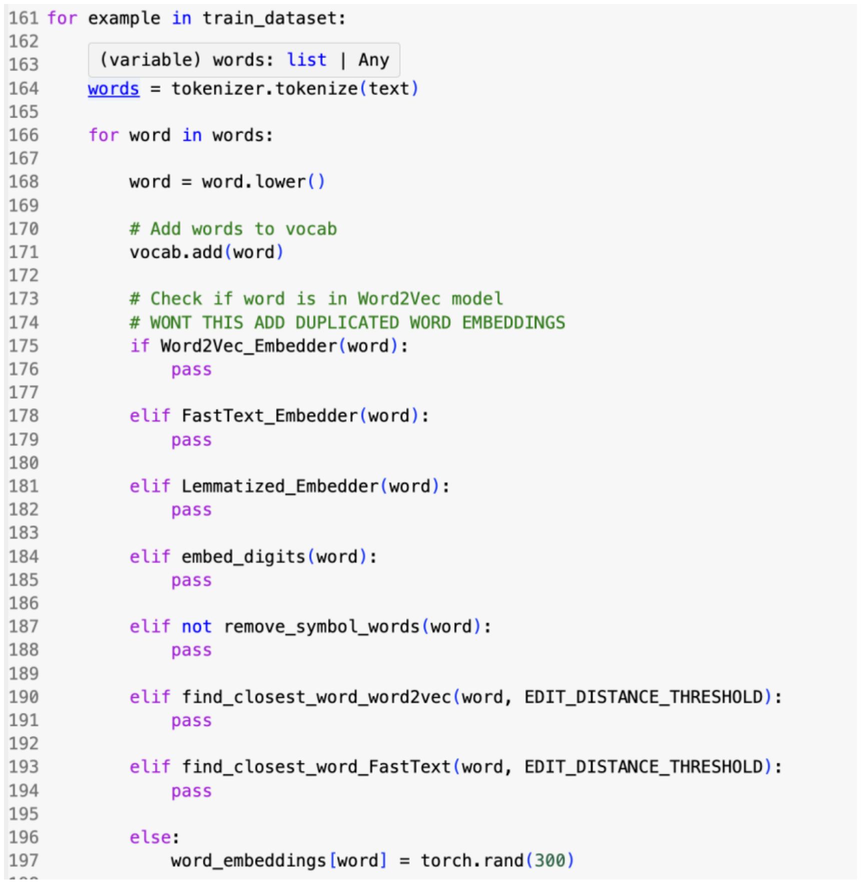
```python
def embed_digits(number):
    check_cycle = len(number)
    counter=0
    digit_list=[]
    for digit in number:
        if digit in number_embeddings:
            digit_list.append(digit)
            counter+=1
            pass
        else:
            oov_words_digit_embedder.add(number)
            return False

    if counter==check_cycle:
        for embeddings in digit_list:
            word_embeddings[embeddings] = number_embeddings[embeddings]
            return True
```

Figure 7: Check whether word is in Word2Vec



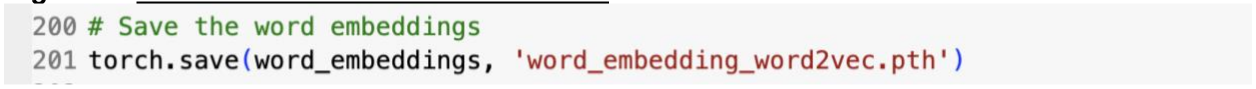
```python
#Function for Lemmatizer
# Lemmatize and see if the word is not in FastText
def Lemmatized_Embedder(word):
    lemma = lemmatizer.lemmatize(word)
    if lemma in word2vec_model:
        word_embeddings[word] = word2vec_model[lemma]
        return True

    elif lemma in fasttext.stoi:
        word_embeddings[word] = fasttext[lemma]
        return True

    else:
        oov_words_lemma.add(word)
        return False
```

Figure 8: Check whether word is in Word2Vec



```python
for example in train_dataset:


    words = tokenizer.tokenize(text)

    for word in words:

        word = word.lower()

        # Add words to vocab
        vocab.add(word)

        # Check if word is in Word2Vec model
        # WONT THIS ADD DUPLICATED WORD EMBEDDINGS
        if Word2Vec_Embedder(word):
            pass

        elif FastText_Embedder(word):
            pass

        elif Lemmatized_Embedder(word):
            pass

        elif embed_digits(word):
            pass

        elif not remove_symbol_words(word):
            pass

        elif find_closest_word_word2vec(word, EDIT_DISTANCE_THRESHOLD):
            pass

        elif find_closest_word_FastText(word, EDIT_DISTANCE_THRESHOLD):
            pass

        else:
            word_embeddings[word] = torch.rand(300)
```

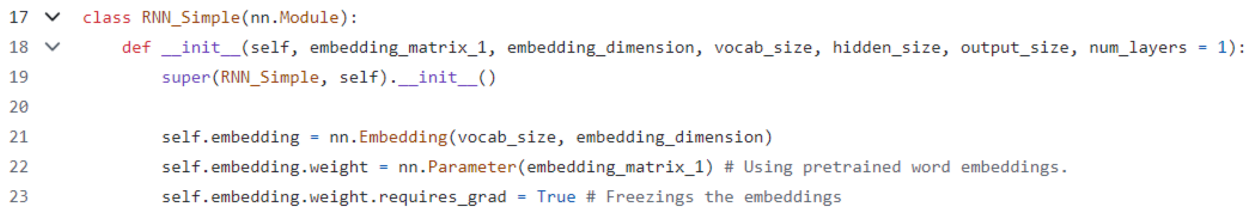
Figure 9: Check whether word is in Word2Vec



```python
# Save the word embeddings
torch.save(word_embeddings, 'word_embedding_word2vec.pth')
```

Figure 10: Saving the embedding file

Q3 b)



```python
class RNN_Simple(nn.Module):
    def __init__(self, embedding_matrix_1, embedding_dimension, vocab_size, hidden_size, output_size, num_layers = 1):
        super(RNN_Simple, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dimension)
        self.embedding.weight = nn.Parameter(embedding_matrix_1) # Using pretrained word embeddings.
        self.embedding.weight.requires_grad = True # Freezings the embeddings
```

Q3 c)

BiLSTM

Introduction and History:

A Bidirectional Long Short-Term Memory (BiLSTM) model is an advanced type of recurrent neural network (RNN) that processes data in both forward and backward directions to capture context from both past and future states. The LSTM itself was introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 to address the vanishing gradient problem in traditional RNNs by using gating mechanisms to control the flow of information. The bidirectional extension of LSTM, known as BiLSTM, was developed to enhance the model's ability to understand context in sequential data, making it particularly useful for tasks such as natural language processing, speech recognition, and time series prediction.

Architecture:

The architecture of a Bidirectional Long Short-Term Memory (BiLSTM) model consists of two LSTM layers that process the input sequence in opposite directions. One LSTM layer processes the sequence from start to end (forward direction), while the other processes it from end to start (backward direction). This dual processing allows the model to capture context from both past and future states, providing a more comprehensive understanding of the sequence. The outputs from both LSTM layers are then concatenated, averaged, or combined in other ways to form the final output, which contains information from both directions.

Each LSTM layer in a BiLSTM model is equipped with gating mechanisms that control the flow of information, helping to retain important features and discard irrelevant ones. These gates include the input gate, forget gate, and output gate, which work together to manage the cell state and hidden state of the LSTM. By processing the sequence in both directions, BiLSTM models can effectively handle long-term dependencies and capture intricate patterns in the data, making them particularly useful for tasks that require a deep understanding of context, such as natural language processing (NLP), speech recognition, and time series prediction.

It is important to note that bi-LSTM produces two hidden representations for each token - which are concatenated - but in the normal model, we specifically concatenate the forward representation of the last_word and the backward representation of the first word because they contain the highest amount of information.

Bi-GRU

History of Bi-GRU

The Bidirectional Gated Recurrent Unit (BiGRU) builds upon the foundation of the Gated Recurrent Unit (GRU), which was introduced in 2014 by Kyunghyun Cho and his colleagues. The

GRU was developed as a simpler alternative to the Long Short-Term Memory (LSTM) network, aiming to address the vanishing gradient problem while reducing computational complexity. Unlike LSTM, GRU combines the forget and input gates into a single update gate and merges the cell state and hidden state, resulting in fewer parameters and faster training times (*Papers With Code - BiGRU Explained*, 2024).

The concept of bidirectional processing in neural networks, which BiGRU employs, was inspired by the need to capture context from both past and future states in sequential data. This approach was particularly beneficial for tasks in natural language processing (NLP), where understanding the full context of a sentence or sequence is crucial. By processing the input data in both forward and backward directions, BiGRU can leverage information from the entire sequence, leading to improved performance in various applications such as speech recognition, machine translation, and sentiment analysis (Dhingra & Cheung, 2021).

Since its introduction, BiGRU has been widely adopted in the research community and has shown competitive performance across different tasks. Its ability to efficiently handle sequential data while maintaining a relatively simple architecture has made it a popular choice for many deep learning practitioners (Onaan, 2022). The ongoing advancements in neural network architectures continue to build on the principles established by GRU and BiGRU, pushing the boundaries of what these models can achieve in real-world applications.

Architecture:
A Bidirectional Gated Recurrent Unit (BiGRU) is a type of recurrent neural network (RNN) that enhances the standard GRU by processing data in both forward and backward directions. This architecture consists of two GRU layers: one that processes the input sequence from start to end (forward direction) and another that processes it from end to start (backward direction). The outputs from both directions are then concatenated, allowing the model to capture context from both past and future states within the sequence. This bidirectional approach is particularly useful for tasks where understanding the full context of the input data is crucial, such as in natural language processing (NLP) applications (Onaan, 2022).

It is important to note that bi-GRU produces two hidden representations for each token - which are concatenated - but in the normal model, we specifically concatenate the forward representation of the last_word and the backward representation of the first word because they contain the highest amount of information. The same also applies to the biLSTSM model described previously.

Q3 e) History of BERT

Let's dive into a little bit of history about BERT:

BERT [Devlin et al., 2018] was originally implemented in the English language at two model sizes:

a) *BERT-base:* 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
b) *BERT-large:* 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.

Both of these models were trained on Toronto BooksCorpus (800M words) and English Wikipedia (2,500M words) and their weights were released on Github

BERT was pre-trained on two tasks, namely:

a) *Masked language modelling (MLM):* In masked language modelling, it randomly masks 15% of input tokens and trains the model to predict them based on the surrounding context. Instead of masking all selected tokens, some are left unchanged or replaced with random words. The reason not all selected tokens are masked is to avoid the dataset shift problem. The dataset shift problem arises when the distribution of inputs seen during training differs significantly from the distribution encountered during inference.

b) Next sentence prediction (NSP): It receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence.