ECE60146: Homework 9

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

In this homework, we performed Natural Language Processing (NLP) where computers learn to understand human language. Our goal was to process text data in a way that computers can comprehend and analyze.

We began by breaking down text into smaller, meaningful chunks through a process called tokenization. This step allows us to convert words or phrases into numerical representations, making it easier for machines to work with them. We explored different levels of tokenization, including word and subword levels. After tokenization, the next step was to extract embeddings, which are numerical representations of words or phrases. These embeddings capture semantic meanings and help our model understand the context of the text data, paving the way for more accurate predictions.

For our main task, we used the DistilBertTokenizer to tackle a sentiment analysis problem. Our objective was to classify text into three sentiments: ”positive,” ”neutral,” and ”negative,” transforming it into a 3-class classification challenge. To achieve this, we implemented a custom Gated Recurrent Unit (GRU) model, a type of Recurrent Neural Network (RNN), using PyTorch's torch.nn.GRU module. Additionally, we experimented with bidirectional RNNs to enhance our model's performance.

2 Methodology

2.1 ”Financial Sentiment Analysis” dataset

2.1.1 Dataset Creation

Like mentioned in previous section, the ”Financial Sentiment Analysis” dataset provided to us was used for sentiment prediction. This dataset comprises three sentiments: ”positive,” ”neutral,” and ”negative.” Consequently, our task transforms into a 3-class classification problem.

The task code reads the dataset from the CSV file and preprocesses it for further analysis. Initially, it initializes empty lists to store sentences and their corresponding sentiments. Then, it opens the CSV file and iterates through each row, extracting sentences and sentiments and appending them to their respective lists.

After collecting the sentences, the code tokenizes them word by word, finding the maximum length among all sentences. It pads the sentences with a '[PAD]' token to ensure they are of equal length. Next, it employs the DistilBertTokenizer from the transformers library to encode the sentences, padding them to the maximum length and truncating if necessary.

Following tokenization, the code constructs a vocabulary mapping to convert tokens to their corresponding token IDs. It assigns a token ID of 0 to the special token '[PAD]' and iterates over each token in the padded sentences, assigning a unique token ID to each token if it is not already present in the vocabulary.

Next, the code converts the tokens to their corresponding token IDs using the constructed vocabulary, generating a new list of lists where each token in each sentence is replaced by its token ID. This preprocessing step prepares the data for subsequent analysis and model training. Finally the DistilBERT model is used to generate word embeddings and subword embeddings from text data. The code loads the pre-trained DistilBERT model and then iterates through each sentence in the dataset, converting the sentence tokens into tensors and passing them through the DistilBERT model to obtain word embeddings. Similarly, the code also processes tokenized sentences to acquire subword embeddings. Both these word and subword embeddings are used in two models each using GRUs and bidirectional-GRUs to perform sentiment analyis.

Each sentence embedding has a size of torch.Size([1, 81, 768]). This means that each embedding consists of a tensor with three dimensions.

1. The first dimension has a size of 1, indicating that there is only one sentence embedded at a time

2. The second dimension has a size of 81, representing the maximum length of the sentence after padding with '[PAD]' tokens. This means that each sentence in the dataset has been padded or truncated to ensure a consistent length of 81 tokens.

3. The third dimension has a size of 768, indicating the dimensionality of the embedding space. Each token in the sentence has been transformed into a vector of length 768, capturing its semantic meaning and context within the sentence.

2.1.2 Dataloader creation

First, the dataset was split into training and testing sets following an 80:20 ratio, as instructed in the task requirements. This was achieved using the train_test_split function from the sklearn.model_selection module. The resulting training set (X_train, y_train) and testing set (X_test, y_test) contain word embeddings and corresponding sentiments.

Next, the sentiments in the training and testing sets were encoded using one-hot encoding to prepare them for training the neural network model. This was done using the OneHotEncoder from the sklearn.preprocessing module. The original sentiment labels were transformed into binary arrays representing each sentiment class.

Following the data preprocessing steps, custom PyTorch dataset classes (CustomDataset) were defined to encapsulate the word embeddings and encoded sentiments. These classes implement the __len__ and __getitem__ methods required by PyTorch datasets.

Additionally, a custom collate function (custom_collate_fn) was implemented to handle batching of data during training. This function stacks the embeddings into tensors and converts the sentiment labels into PyTorch tensors.

Finally, custom data loaders (train_loader and test_loader) were created using PyTorch's DataLoader class, which allows for efficient loading of data in batches during training and testing. These data loaders utilize the custom dataset classes and collate function to provide batches of data for training and evaluation.

```
    import torch
from torch.utils.data import Dataset, DataLoader
# Step 1: training set (X\_train, y\_train) and testing set (X\_test, y\_test)
                                         contain word embeddings and corresponding
                                          sentiments were created
# Step 2: Define a custom dataset class
class CustomDataset(Dataset):
    def __init__(self, word_embeddings, sentiments):
        self.word_embeddings = word_embeddings
        self.sentiments = sentiments
    def __len__(self):
        return len(self.word_embeddings)

    def __getitem__(self, idx):
        return self.word_embeddings[idx], self.sentiments[idx]

# Step 3: Implement the custom collate function to handle batching
def custom_collate_fn(batch):
    embeddings, sentiments = zip(*batch)
    embeddings = torch.stack([torch.squeeze(embedding) for embedding in embeddings
                                          ])  # Remove dimensions with size 1
    sentiments = torch.tensor(sentiments)  # Convert list to tensor with integer
                                          type
    return embeddings, sentiments

# Step 4: Create custom data loaders for the train and test sets
train_dataset = CustomDataset(X_train, y_train_encoded)
test_dataset = CustomDataset(X_test, y_test_encoded)

# Create custom data loaders for the train and test sets
train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True, collate_fn=
                                          custom_collate_fn)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False, collate_fn=
                                          custom_collate_fn)
```

2.1.3 Network, Training and Testing Architecture

A custom GRU network and bidirectional GRU were implemented using nn.GRU from PyTorch. These networks were inspired by previous year's code [2], which, in turn, was derived from DLStudio code [1].

Custom GRU and Bidirectional GRU Networks The custom GRU network, *GRUnetWithEmbeddings* and Bidirectional GRU *NetworksBiGRUnetWithEmbeddings* is designed for sentiment analysis tasks. The total number of parameters are about 1.8M for the former and twice that for the latter. It takes word embeddings as input and predicts the sentiment of the input text. The network architecture consists of a GRU layer (3 of them with 100 hidden states each) followed by a fully connected layer (3 node for the main task while 2 nodes for the extra credit) . The GRU layer processes the input embeddings to capture the sequential information, and the fully connected layer produces the final output. An optional dropout layer is included in the GRU layer for regularization. ReLU activation is applied between the GRU and fully connected layers. Log softmax activation is used for the output layer in the extra credit task while it is skipped in the main task. This is because the training for the main task uses nn.CrossEntropyLoss() in training which has an Log softmax activation already built into it. In the extra credit task I use Log softmax activation and in training loop I use nn.NLLLoss.

```
import torch.nn as nn

class GRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(GRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        # Define GRU layer
        self.gru = nn.GRU(input_size, hidden_size, num_layers,dropout=0.2)

        # Define fully connected layer
        self.fc = nn.Linear(hidden_size, output_size)

        # Define activation function
        self.relu = nn.ReLU()

    def forward(self, x, h):
        # Forward pass through GRU layer
        out, h = self.gru(x, h)

        # Apply ReLU activation function
        out = self.fc(self.relu(out[-1]))

        return out, h

    def init_hidden(self):
        # Initialize hidden state with zeros
        weight = next(self.parameters()).data
        hidden = weight.new(self.num_layers, 1, self.hidden_size).zero_()
        return hidden
##############################################################################
# Bidirectional torch.nn GRU with Embeddings
class BiGRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(BiGRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers, bidirectional=True,
                                                  dropout=0.2)
        self.fc = nn.Linear(2*hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.fc(self.relu(out[-1]))
        return out, h

    def init_hidden(self):
        weight = next(self.parameters()).data
        #                  num_layers  batch_size    hidden_size
        hidden = weight.new(  2*self.num_layers,          1,         self.
                                                  hidden_size    ).zero_()
        return hidden
```

Training and Testing Training and testing logics are very similar to a usual neural network training and testing ideas. In the training routine, the GRU network is trained on the provided dataset using cross-entropy loss. This process involves iterating through the dataset for multiple epochs while optimizing the network parameters using the Adam optimizer. The Adam optimizer is configured with a learning rate of 1e-4 and momentum parameters (betas) set to (0.8, 0.999). Throughout training, progress is displayed periodically, indicating the current loss. Once training is complete, the trained model is saved, and a plot depicting the training loss versus iterations is generated.

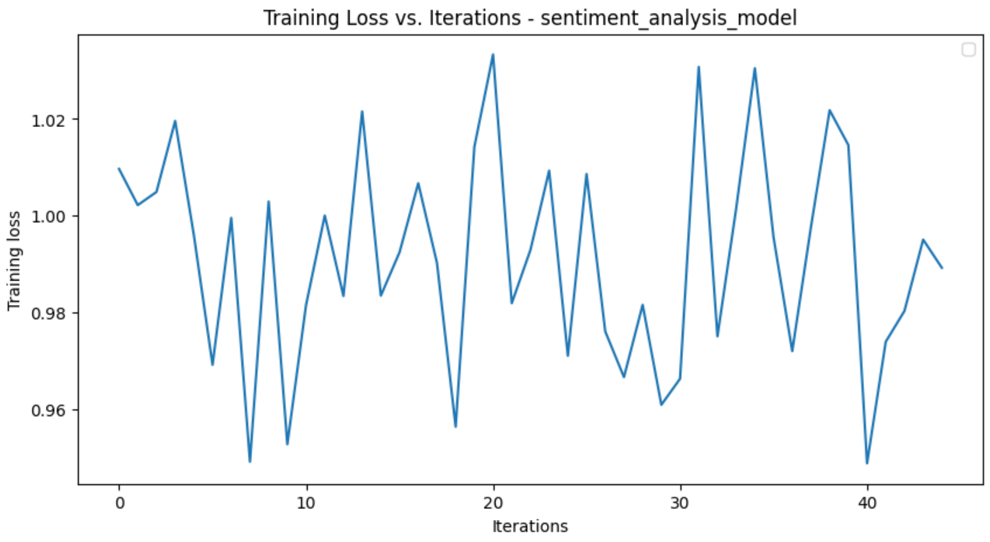

Figure 1: Loss curve of GRU with Word Embeddings

The validation routine evaluates the trained model on a separate validation dataset to assess its performance. It loads the trained model weights, performs inference on the validation dataset, and calculates metrics such as overall classification accuracy and confusion matrix. The confusion matrix provides insights into the model's performance across different sentiment classes. Additionally, a visualization of the confusion matrix is generated to aid in result interpretation.

2.1.4 Outputs

The X-axis in the loss curve contains the loss values for iterations, where each iteration represents a single step in the training process. However, to condense the plot and provide a clearer visualization, the X-axis labels are divided by 10 over all the epochs. This division helps to scale down the number of iterations displayed on the plot, making it easier to interpret while still capturing the overall training progress across epochs.

GRU with Word Embeddings The training loss and Confusion Matrix over 5 epochs is shown in Figures 1 and 2

Bidirectional GRU with Word Embeddings The training loss and Confusion Matrix over 5 epochs is shown in Figures 3 and 4

The loss trend over epochs and iterations reveals an erratic pattern for both the GRU and bidirectional GRU models trained on word embeddings. Despite training over multiple epochs, the loss curve does not exhibit a consistent decrease, indicating difficulties in convergence and model optimization. Consequently, this suboptimal training behavior translates into poor predictive performance, with an overall accuracy of only 53.21% for the GRU model and 53.29% for the bidirectional GRU model.

Upon closer examination, it becomes evident that the predictions are predominantly skewed towards the neutral sentiment class. This observation aligns with the imbalanced nature of the dataset, where the majority of samples belong to the neutral sentiment category (we have an imbalanced dataset with positive, neutral and negative sentences numbered 860,3130, and 1852

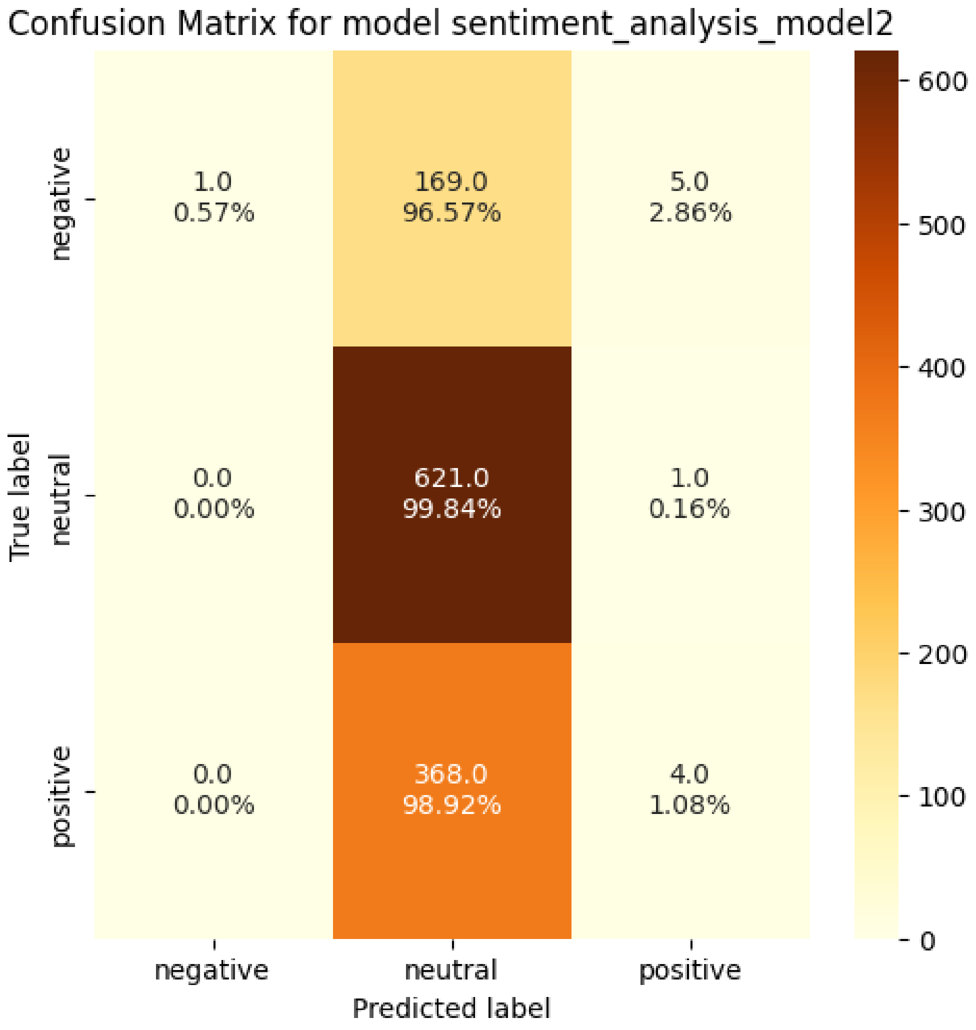

Figure 2: Confusion Matrix for GRU with Word Embeddings

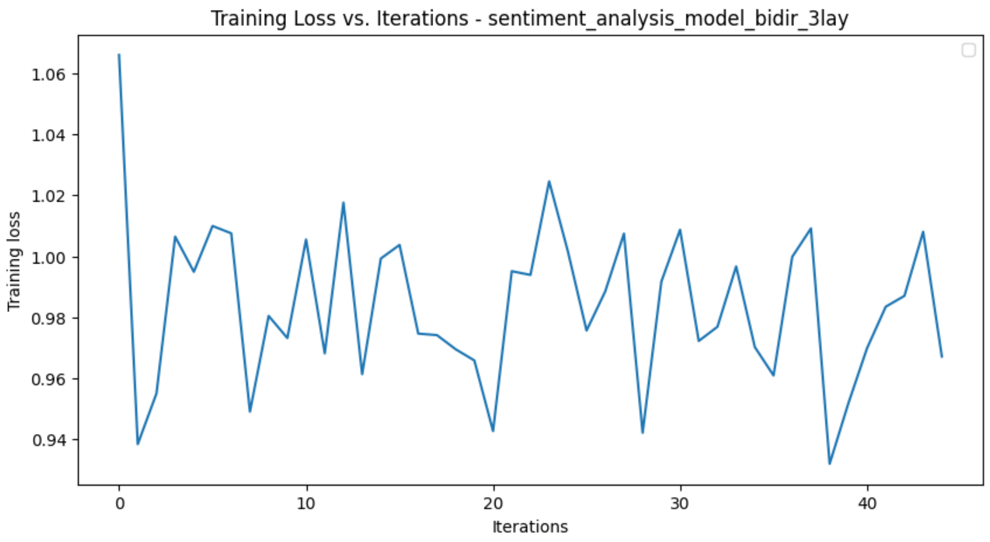

Figure 3: Loss curve of Bidirectional GRU with Word Embeddings

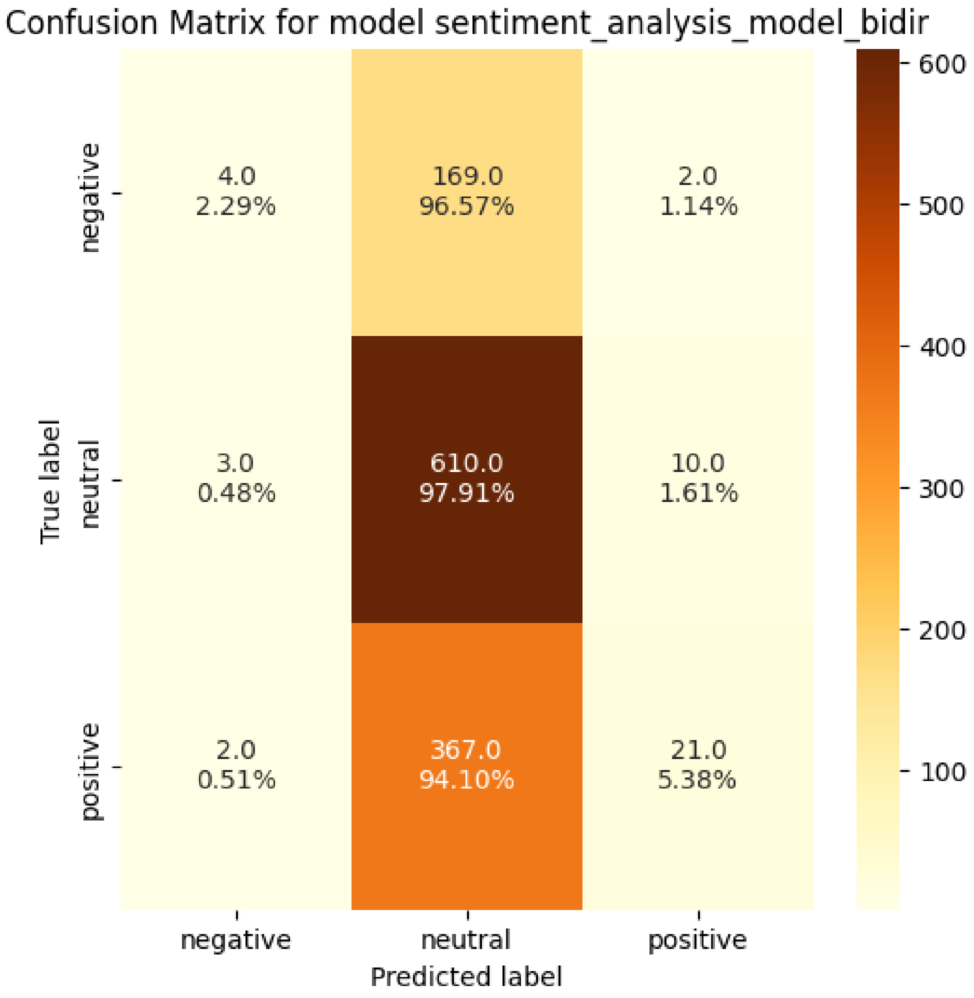

Figure 4: Confusion Matrix for Bidirectional GRU with Word Embeddings

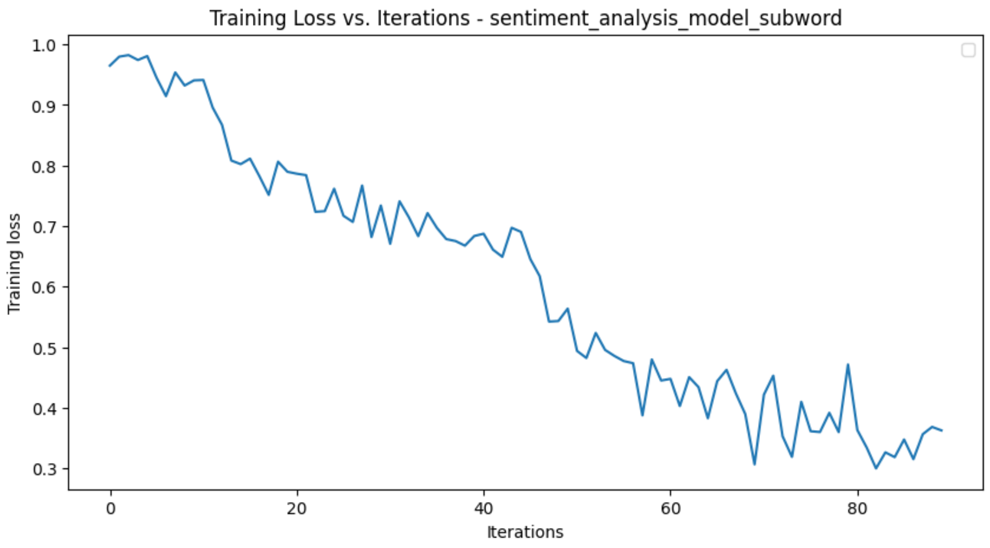

Figure 5: Loss curve of GRU with Subword Embeddings

respectively). Despite the presence of positive and negative sentiment samples, their relatively smaller proportions within the dataset do not significantly influence the model's predictions. Infact the prediction accuracy is insufficient as we can just have network predict everything as neutral and achieve that much accuracy.

This challenge underscores the limitations of the current approach, as the models struggle to discern between the different sentiment classes effectively. Consequently, the models tend to generalize towards the most prevalent sentiment class (neutral), resulting in subpar accuracy levels. To address this issue and improve predictive performance, alternative approaches are warranted. This issue is resolved when we use subword embeddings

GRU with Subword Embeddings The training loss and Confusion Matrix over 10 epochs is shown in Figures 5 and 6

Bidirectional GRU with Subword Embeddings The training loss and Confusion Matrix over 15 epochs is shown in Figures 7 and 8

For both GRU and bidirectional GRU with the subword embeddings, the loss trend depicted in the provided logs demonstrates a notable improvement compared to the word embeddings case. Across epochs and iterations, the loss steadily decreases, indicating effective optimization and convergence of the models. This progressive reduction in loss values suggests that the models are learning from the training data and improving their predictive capabilities over time.

As a result of this improved training behavior, the overall accuracy of the models also shows significant enhancement. In the case of the GRU model, the accuracy reaches 77.84%, while the bidirectional GRU model achieves an accuracy of 76.39%. This notable increase in accuracy indicates that the models are better able to differentiate between the different sentiment classes within the dataset.

One possible reason for this improvement could be attributed to the utilization of subword embeddings instead of word embeddings. Subword embeddings provide a more nuanced representation of text, capturing morphological and contextual information more effectively. This enhanced representation allows the models to better understand and differentiate between the subtle nuances

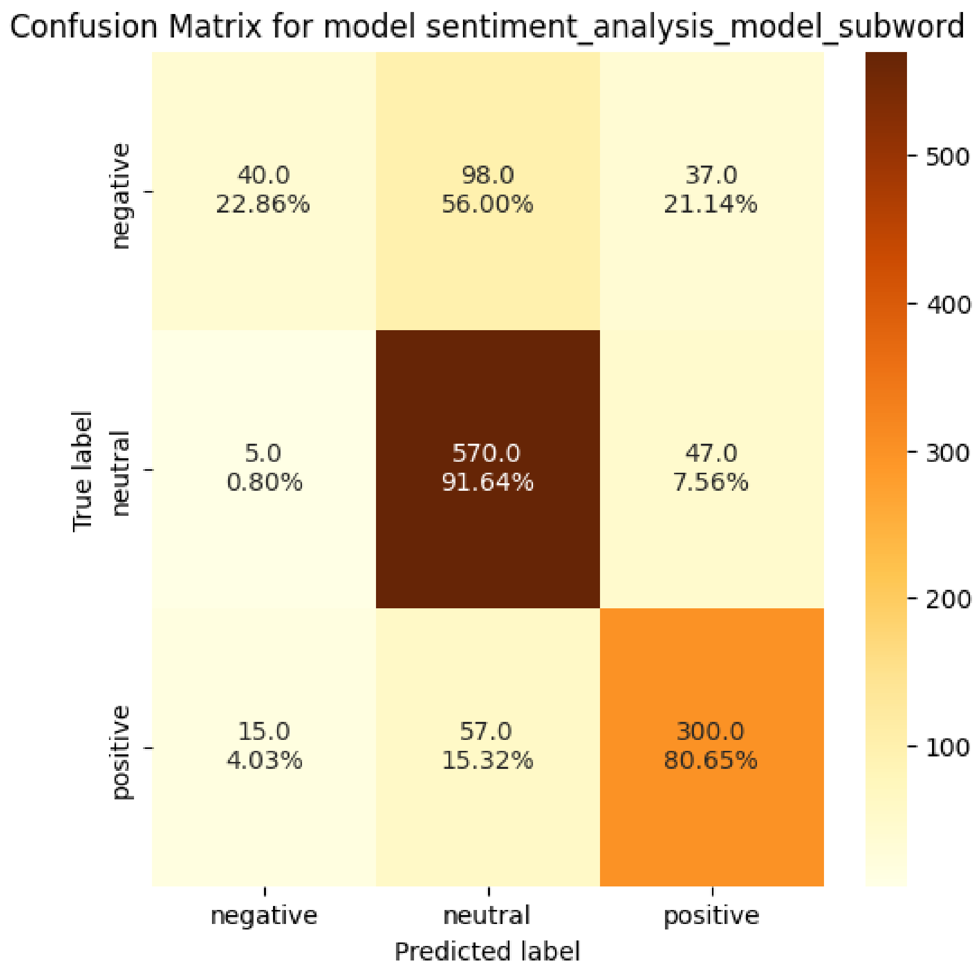

Figure 6: Confusion Matrix for GRU with Subword Embeddings

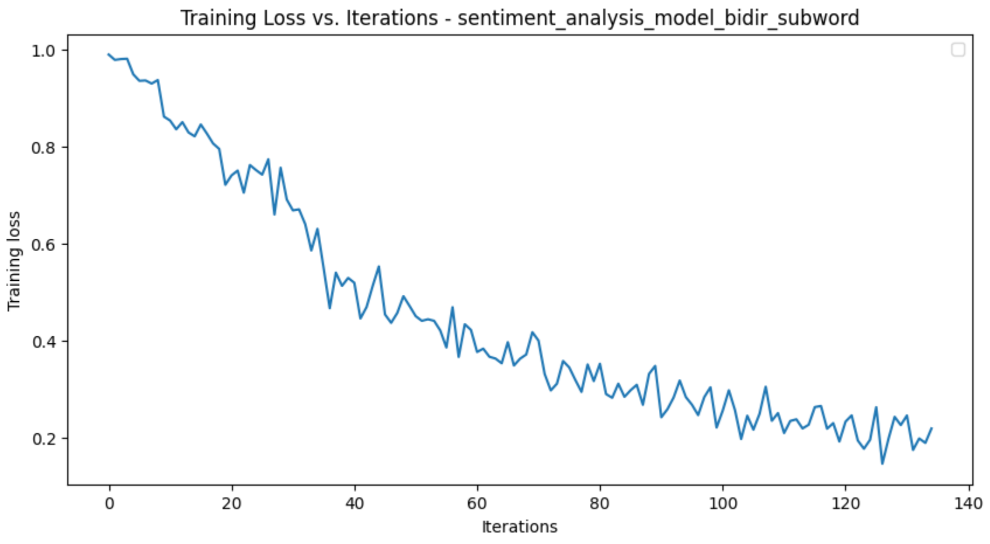

Figure 7: Loss curve of Bidirectional GRU with Subword Embeddings

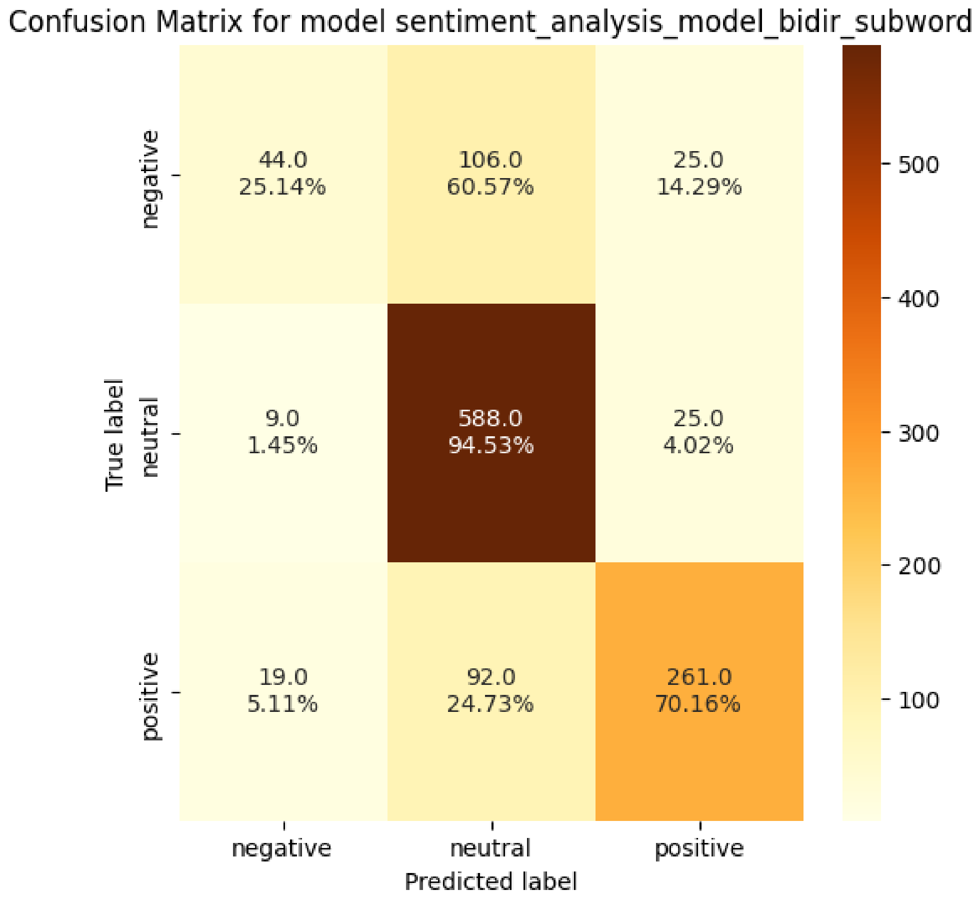

Figure 8: Confusion Matrix for Bidirectional GRU with Subword Embeddings

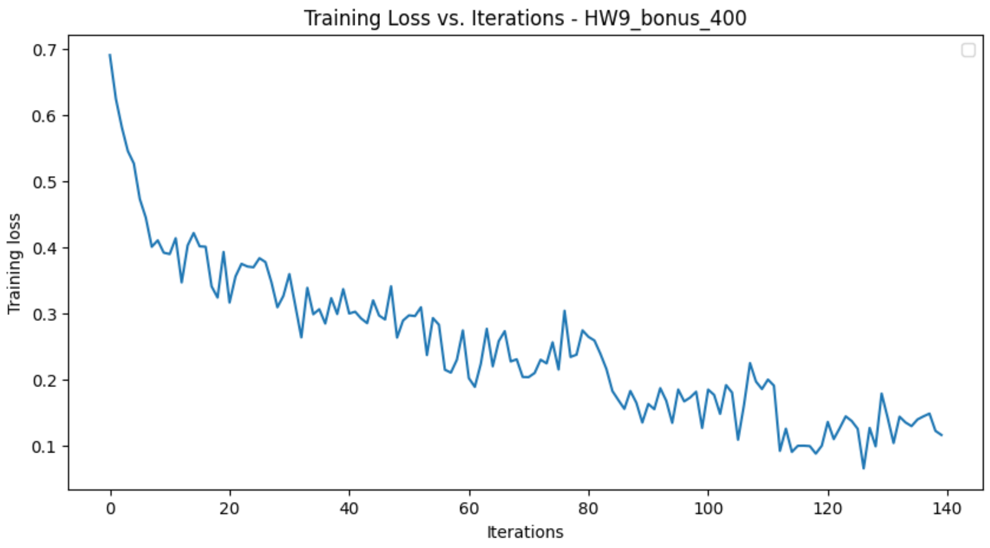

Figure 9: Loss curve of GRU for sentiment_dataset_train_400.tar.gz dataset

of sentiment expressed in the text, leading to improved predictive performance.

2.2 "Extra Credit" Task

For the extra credit task, sentiment analysis was conducted on the text classification dataset provided on the course website. The task involved using the same unidirectional and bidirectional GRUs as in the main task, with the only difference being the output layer, which consisted of only two nodes due to the binary nature of the classification problem. Additionally, the log softmax activation function along with the nn.NLLoss (negative log likelihood loss) was employed for training.

Two different datasets were utilized for training: sentiment_dataset_train_400.tar.gz and sentiment_dataset_train_200.tar.gz (as well as their test counterparts for testing). The former dataset had a vocabulary size of 64,350, while the latter had a vocabulary size of 43,285. These datasets varied in size and vocabulary, potentially influencing the model's learning capabilities and performance.

Using a similar approach as that of the main task, a custom GRU class was created and used in the GRU and bidirectional GRU network for classifying positive and negative reviews. Same training routine was reused to train and log the losses at periodic checkpoints.

A SentimentAnalysisDataset class was designed for this sentiment analysis task based on DLStudio Code. It processes text datasets *sentiment_dataset_train_{400.tar.gz and 200.tar.gz}* (and their test counterparts), converts it into tensors suitable for deep learning models, and provides methods to retrieve review embeddings and sentiment labels (positive vs negative). The class initializes by reading the dataset file and loading word embeddings (word2vec). It converts words into embeddings (torch.Size([1, 54, 300]) and sentiments into one-hot encoded tensors (torch.Size([1, 2])). Overall, it facilitates the preparation of training and testing data for sentiment analysis models.

sentiment_dataset_train_400.tar.gz dataset with GRU The training loss and Confusion Matrix over 5 epochs is shown in Figures 9 and 10.

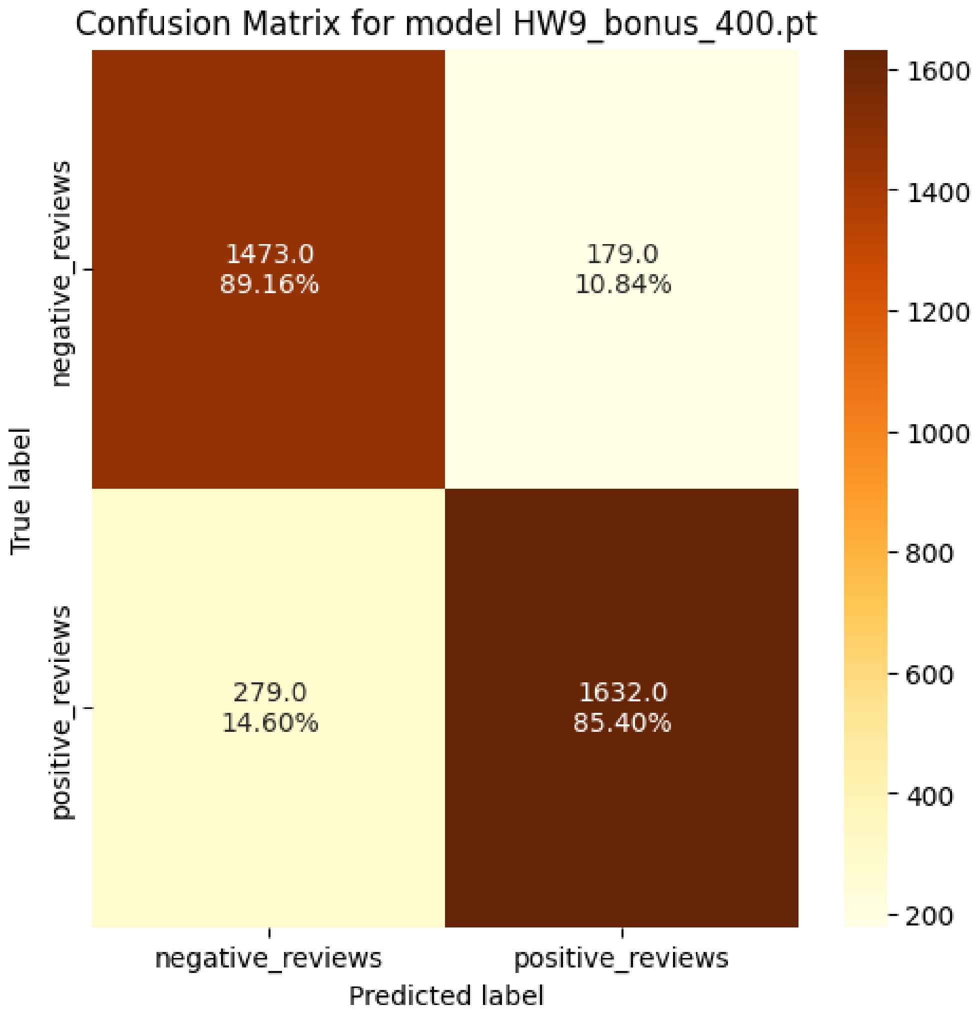

Figure 10: Confusion Matrix GRU of sentiment_dataset_train_400.tar.gz dataset

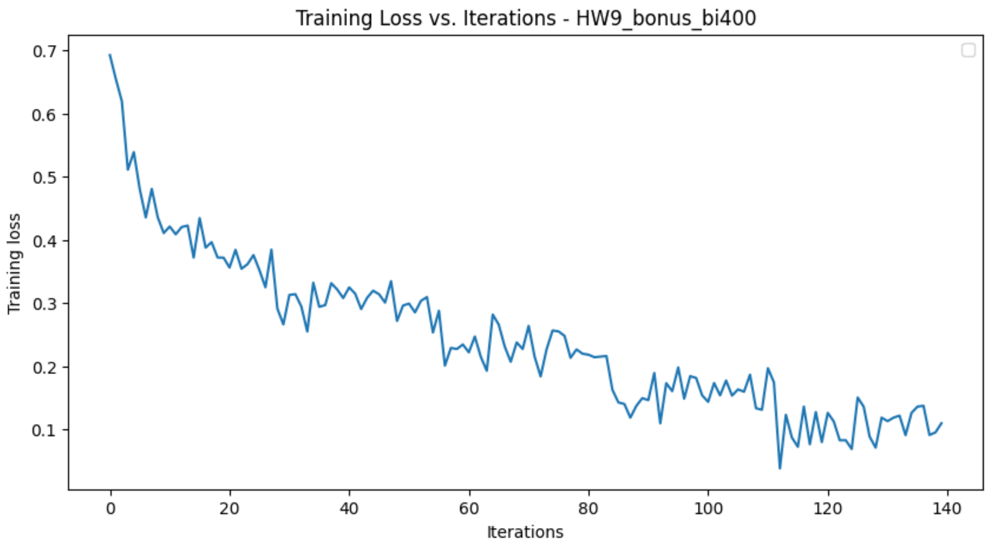

Figure 11: Loss curve of Bidirectional GRU for sentiment_dataset_train_400.tar.gz dataset

sentiment_dataset_train_400.tar.gz dataset with Bidirectional GRU The training loss and Confusion Matrix over 5 epochs is shown in Figures 11 and 12.

Both the GRU and bidirectional GRU loss trends shows a steady decrease over the epochs and iterations. Initially, the loss is relatively high, but it progressively decreases as the training progresses. This indicates that the model is effectively learning from the data and improving its predictive performance. Additionally, the elapsed time increases with each epoch, reflecting the computational effort required for training. Overall, the trend suggests successful training and learning by the model.Due to this in the experiments, I observed an impressive accuracy of 87.17% for the unidirectional GRU model and 86.64% for the bidirectional GRU model. Interestingly, despite having a higher number of vocabulary words in this experiment compared to our main task, we achieved better performance.

sentiment_dataset_train_200.tar.gz dataset with GRU The training loss and Confusion Matrix over 5 epochs is shown in Figures 13 and 14.

sentiment_dataset_train_200.tar.gz dataset with Bidirectional GRU The training loss and Confusion Matrix over 5 epochs is shown in Figures 15 and 16.

Once again an impressive accuracy of 87.95% for the unidirectional GRU and 86.30% for bidirectional GRU case was observed. One possible reason for this remarkable performace compared to the main task could be that we used word embeddings generated by word2vec for this task, specifically the 'word2vec-google-news-300' model, which is known for its comprehensive coverage and high-quality embeddings.

Another possible reason for this improved performance could be attributed to the nature of the dataset itself. Unlike our main task, which likely involved a more complex classification problem with three classes, the sentiment analysis dataset used in this experiment was a binary classification task, involving only two classes. This simplified task structure may have made it easier for the models to discern patterns and make accurate predictions.

Despite the sentiment analysis dataset having a higher number of vocabulary items compared to our main task, the utilization of pre-trained word embeddings from word2vec might have con-

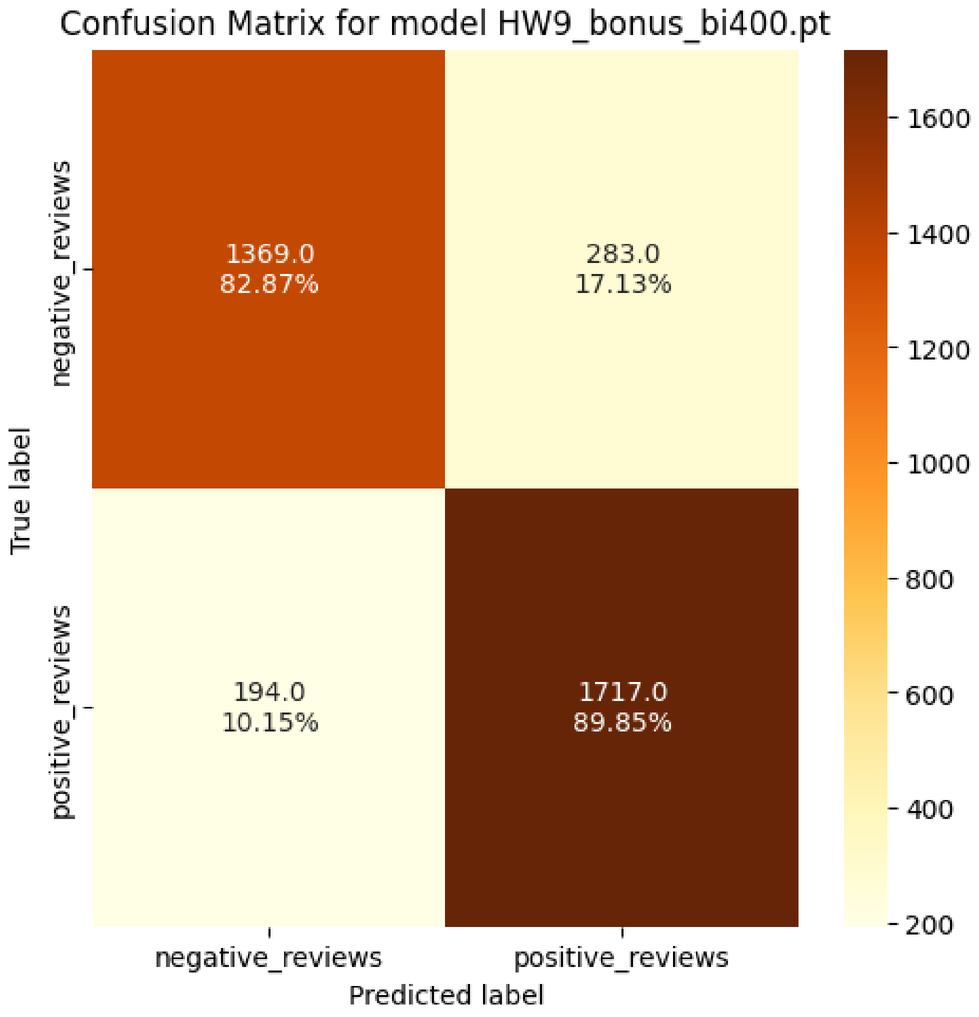

Figure 12: Confusion Matrix Bidirectional GRU of sentiment_dataset_train_400.tar.gz dataset

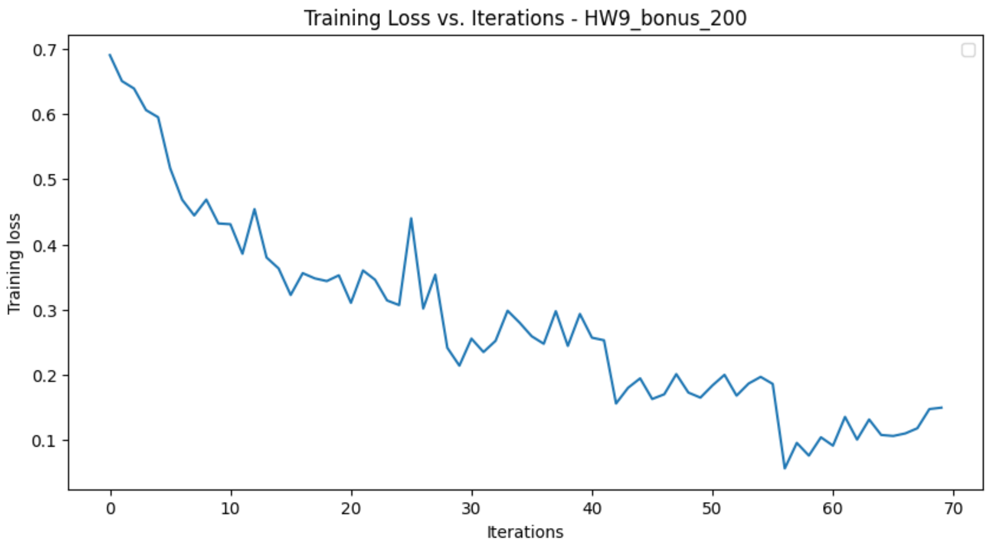

Figure 13: Loss curve of GRU for sentiment_dataset_train_200.tar.gz dataset

tributed to the enhanced performance. Word embeddings capture semantic relationships between words, allowing the model to better understand the underlying meaning and context of the text. By leveraging these pre-trained embeddings, the models may have been able to generalize more effectively to unseen data and extract meaningful features for sentiment analysis.

Overall, the combination of a simplified classification task, along with the utilization of high-quality word embeddings, likely played a significant role in achieving the impressive accuracy results observed in this experiment.

2.3 Code

Includes two Python files containing code for both the main task and the extra credit task are available in the submission directory provided (skipped appending here due to length).

2.4 Conclusion

In summary, our study on sentiment analysis using RNN models like unidirectional GRU and bidirectional GRU showed positive outcomes. By using pre-trained word embeddings like 'word2vec-google-news-300', we boosted the model's performance, even though the sentiment dataset had a larger vocabulary. These results emphasize the effectiveness of RNNs and pre-trained word embeddings in analyzing sentiments from text and their capability to understand text and detect sentiment accurately.

References

[1] `https://engineering.purdue.edu/kak/distDLS/`

[2] `https://github.com/souradipp76/Deep-Learning-ECE60146/tree/main/hw8`

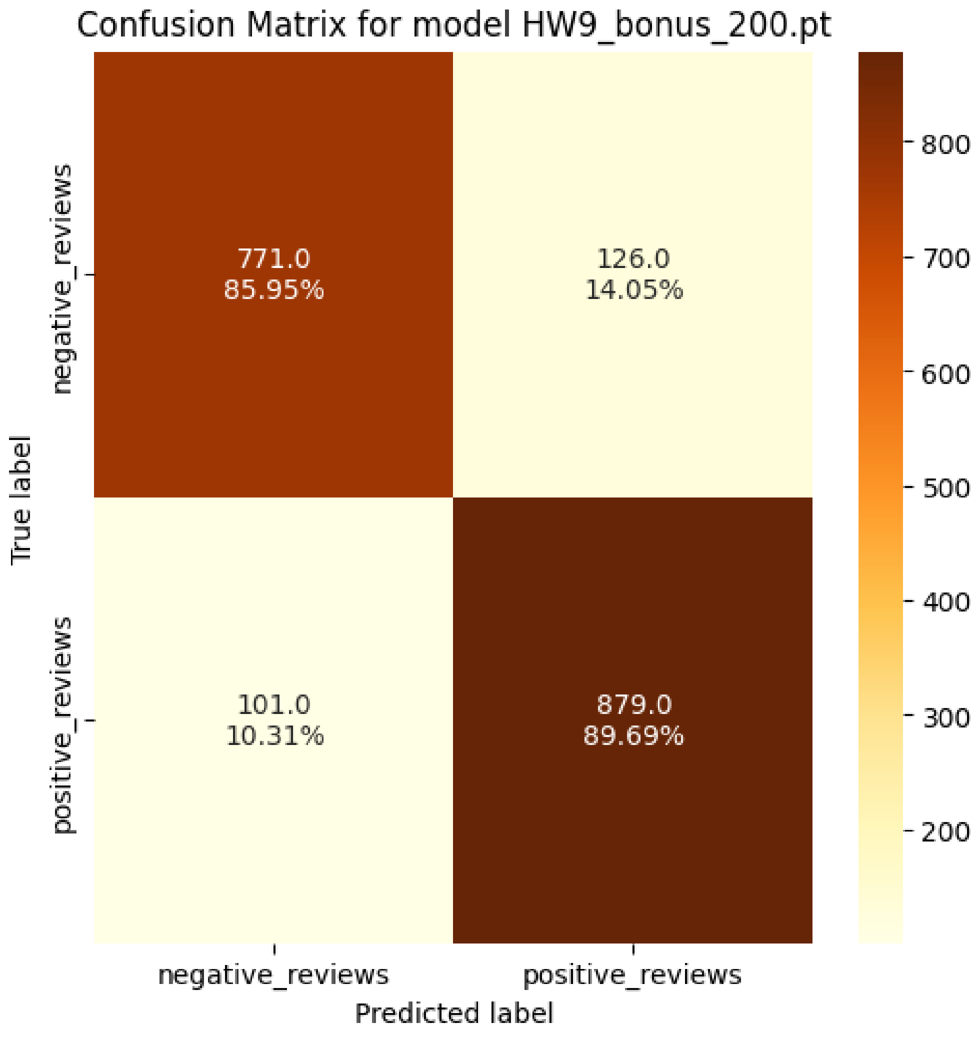

Figure 14: Confusion Matrix GRU of sentiment_dataset_train_200.tar.gz dataset

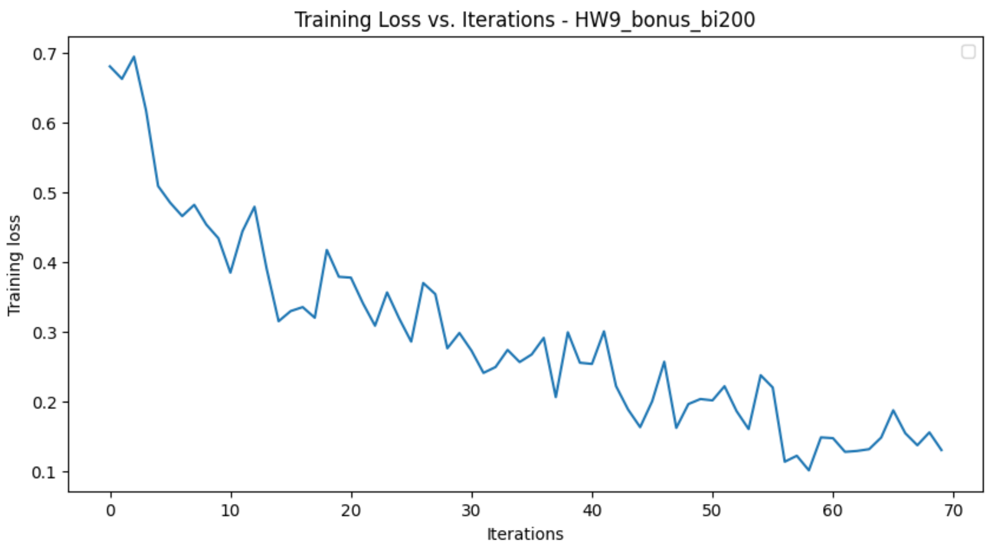

Figure 15: Loss curve of Bidirectional GRU for sentiment_dataset_train_200.tar.gz dataset

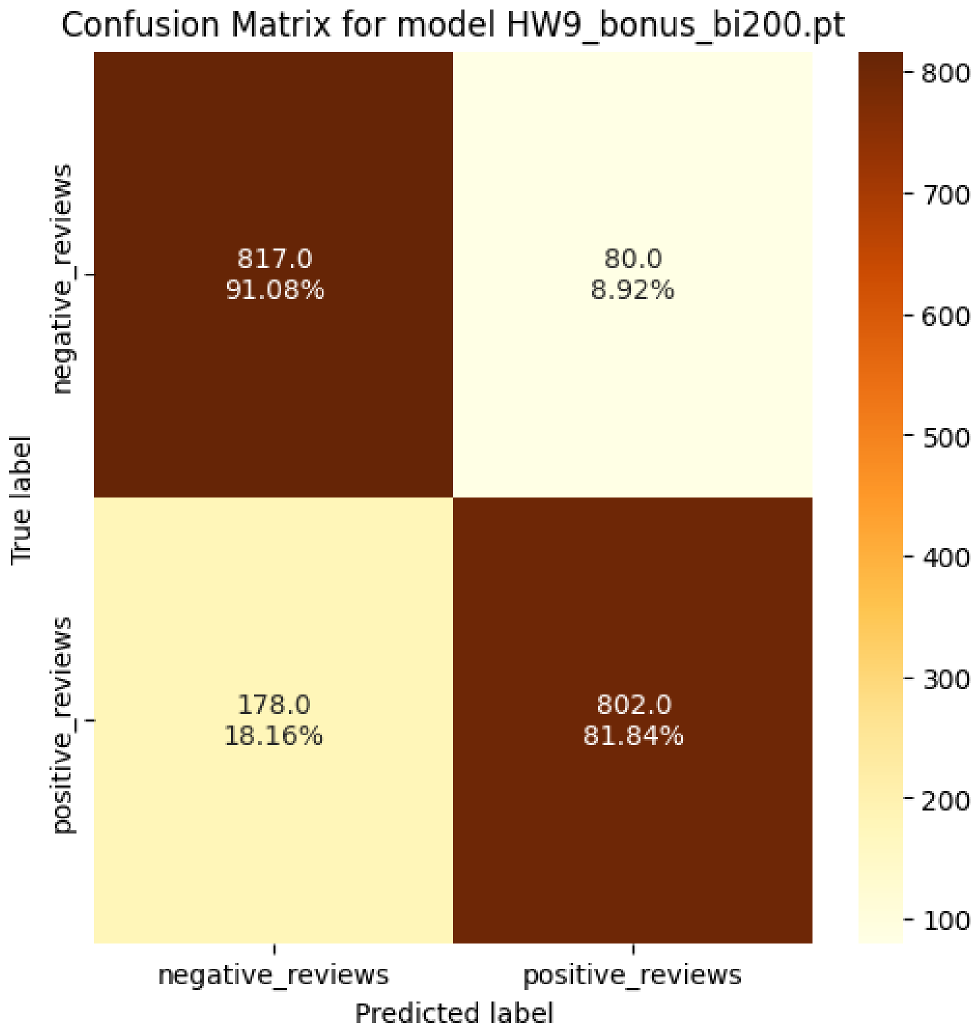

Figure 16: Confusion Matrix Bidirectional GRU of sentiment_dataset_train_200.tar.gz dataset