RNN using GRU and Bidirectional GRU for Sentiment Analysis and Classification

Introduction:

The focus of this homework is the use of Gated Recurrent Units (GRUs) along with Recurrent Neural Networks (RNNs) to solve the vanishing gradient problem and use it to classify sentiments. This is followed with wrod2vec word embedding, which is used to separate reviews from a given dataset into positive and negative categories.

RNN:

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed to effectively handle sequential data by capturing dependencies and patterns over time. Unlike traditional feedforward neural networks, RNNs possess an internal memory mechanism that enables them to retain information about previous inputs while processing the current input. This memory feature allows RNNs to exhibit dynamic temporal behavior, making them particularly suitable for tasks such as natural language processing, time series prediction, and speech recognition.

An RNN is fundamentally made up of a network of interconnected units, or cells, where each cell has its own internal state and receives an input. The network is then able to maintain information over time by updating this state in response to both the input at that point and the data that was stored in the previous state.

GRU

However, traditional RNNs can suffer from the vanishing gradient problem, where gradients diminish exponentially as they propagate backward in time, limiting the network's ability to learn long-term dependencies. To address this issue, various extensions of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), have been developed, as they introduce long-term memory

into the network. GRUs use gated mechanism, which consists of two gates: the update gate and the reset gate. These gates control the flow of information within the network, allowing GRUs to selectively update the hidden state and regulate the information flow across time steps.

The update gate determines how much of the previous state should be retained and how much of the new state should be incorporated, while the reset gate controls the amount of past information to forget. In the below formulas:

x is the inputs, and h is the hidden state of x.
Binary valuation is achieved by a sigmoid activation function.

The states are updated using the equation below:

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1}) \\
\mathbf{r}_t &= \sigma(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1}) \\
\tilde{\mathbf{h}}_t &= tanh(W_h \mathbf{x}_t + U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

Network and Training routine:

All networks were run for 10 epochs, batch size of 1, input size of 768, hidden size of 100, output size of 3 (as we classified based on 3 review classes neutral, positive or negative), learning rate 1e-4, and trained used the Adam optimizer(betas = (0.8, 0.999)).

Unidirectional GRU (GRUnet) Architecture used:

- **Initialization**: The GRUnet initializes with the parameters `input_size`, `hidden_size`, `output_size`, and `num_layers`. It sets up a GRU layer using `nn.GRU` with the specified input size, hidden size, and number of layers.
- **Forward Pass**: During the forward pass, the input sequence `x` is passed through the GRU layer. The output of the GRU layer (`out`) and the hidden state (`h`) are returned. The last output of the sequence (`out[:, -1]`) is fed into a fully

connected layer (`nn.Linear`) followed by a ReLU activation (`nn.ReLU`). Finally, a LogSoftmax activation (`nn.LogSoftmax`) is applied to obtain the output probabilities.

- **Hidden State Initialization**: The `init_hidden` method initializes the hidden state of the GRU with zeros. It creates a tensor with dimensions `(num_layers, 1, hidden_size)` **,where 1 is the batch size.**

```
class GRUnet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size,
num_layers, drop_prob=0.2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layer
s)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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

Bidirectional GRU (BiGRUnet) Architecture used:

- The `__init__` method initializes the parameters of the network. It takes input parameters such as `input_size` (dimensionality of input features), `hidden_size` (number of hidden units in each RNN layer), `output_size` (dimensionality of the output), `num_layers` (number of RNN layers), and `drop_prob` (dropout probability for regularization).
- **GRU Layer**: The core of the architecture is the Gated Recurrent Unit (GRU) layer, defined by `self.gru`. This layer processes input sequences `x` and maintains hidden states `h`. It has `num_layers` layers, each with `hidden_size` hidden units. By setting `bidirectional=True`, the GRU is made bidirectional, meaning it processes sequences both forwards and backwards, effectively capturing information from both past and future contexts.
- The fully connected layer (`self.fc`) linearly transforms the concatenated output of the forward and backward GRU passes to the output size. The ReLU activation function (`self.relu`) is applied to the output of the fully connected layer to introduce non-linearity to the model.
- **Output Layer**: The output of the ReLU activation is passed through a log softmax layer (`self.logsoftmax`). This layer converts the raw output scores into log probabilities, facilitating training with logarithmic loss functions like cross-entropy loss.
- **Forward Pass**: The `forward` method defines the forward pass of the network. Given an input sequence `x` and an initial hidden state `h`, it processes the input sequence through the bidirectional GRU layer, concatenates the output from the forward and backward passes, applies the fully connected layer, ReLU activation, and log softmax to produce the final output probabilities.
- **Hidden State Initialization**: The `init_hidden` method initializes the hidden state `h` for the first time step of the input sequence. It creates a tensor of zeros with the appropriate dimensions based on the batch size (`batch_size`**=1**), number of layers, and whether the network is bidirectional. This method is typically called before the forward pass at the start of each new sequence.

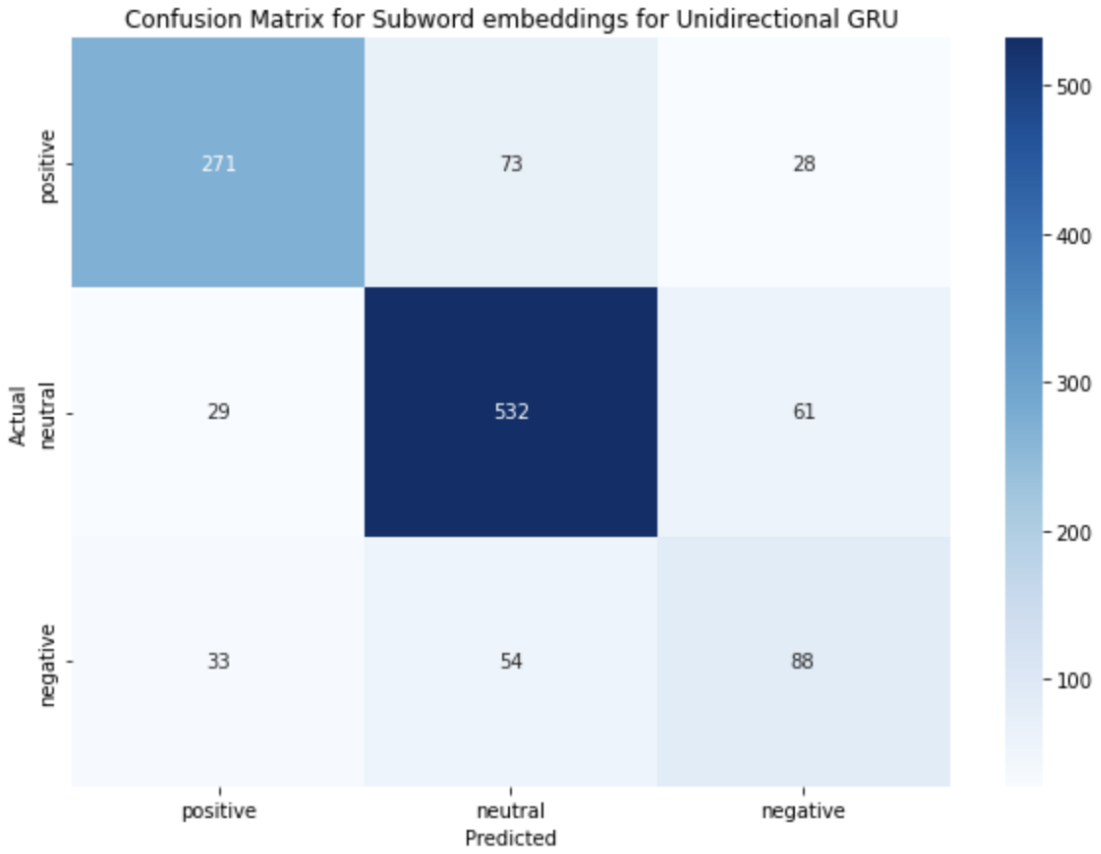
```
class BiGRUnet(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, output_size,
num_layers, drop_prob=0.2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = True

        self.gru = nn.GRU(input_size, hidden_size, num_layer
s, dropout=drop_prob, bidirectional=True) # True for bidirect
ional GRU
        self.fc = nn.Linear(hidden_size * 2, output_size)  #
Multiply by 2 for bidirectionality
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = torch.cat((out[:, -1, :self.hidden_size], out
[:, 0, self.hidden_size:]), dim=1)
        out = self.fc(self.relu(out))
        out = self.logsoftmax(out)
        return out, h

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

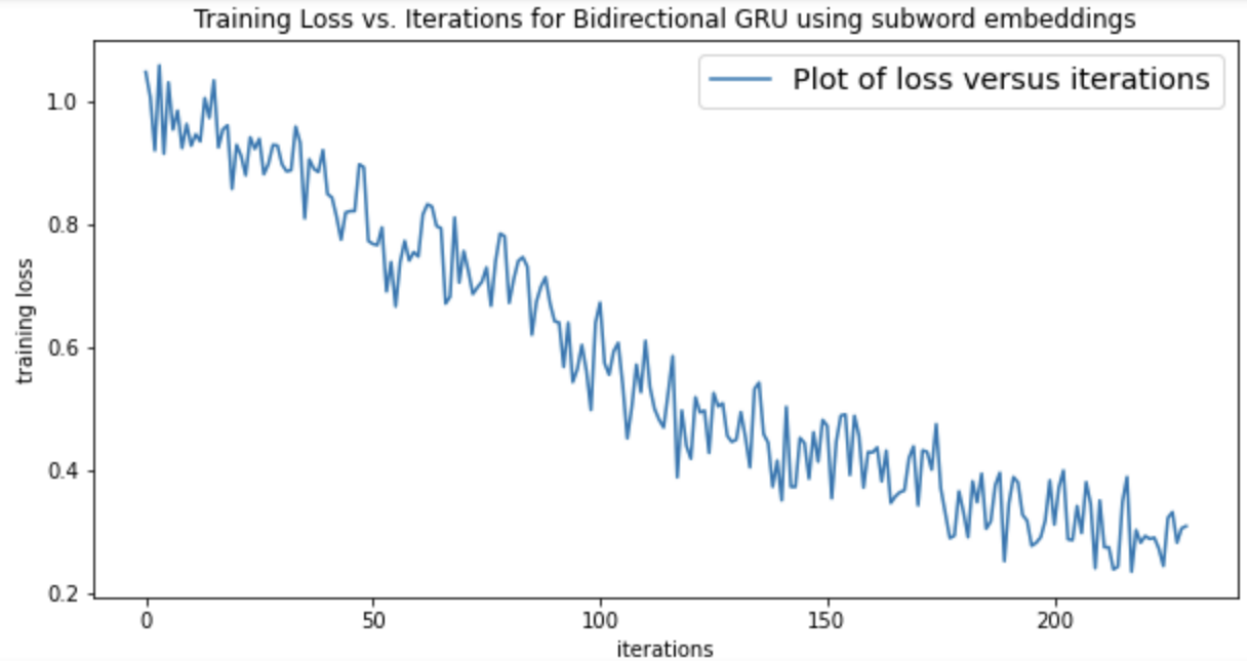
Conclusion and Test Performance plots:

- Loss plot, classification accuracy and confusion matrix for Unidirectional GRU network using subword embeddings.

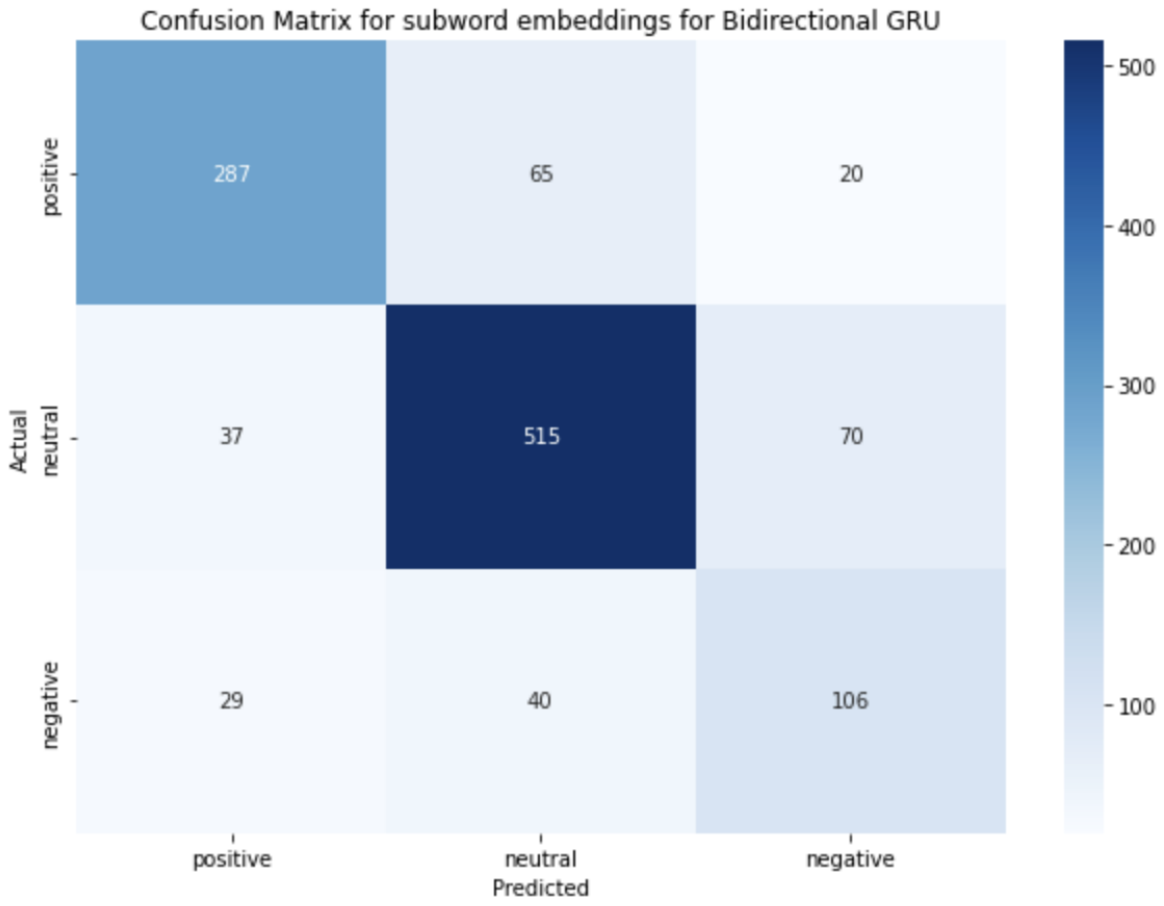

```
Overall classification accuracy: 76.22%
```



- Loss plot, classification accuracy and confusion matrix for Bidirectional GRU network using subword embeddings.




```
Overall classification accuracy: 77.67%
```



Does using a bidirectional scan make a difference in terms of test performance?

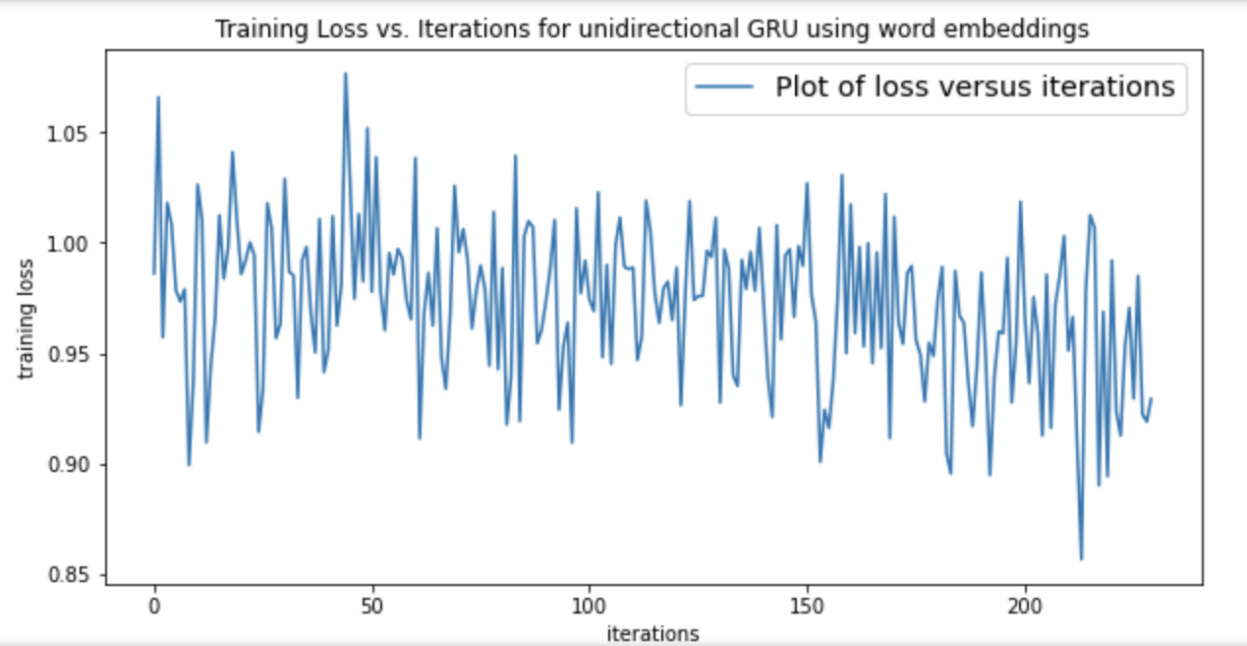
Yes, using a bidirectional scan made a difference in terms of test performance, in the above experiment. As can be observed when the Unidirectional GRU network using subword embeddings is used, the accuracy achieved is `76.22%`, while when the Bidirectional GRU network using subword embeddings is used, the accuracy achieved is `77.67%,` that means the accuracy is improved and when using Bidirectional Architecture. Also from the confusion matrix it can be observed that the classification is more clear between different classes and we can see better classified number of test samples in the Confusion matrix formed using the Bidirectional architecture.

Comparison of the test performances of the both RNN implementations

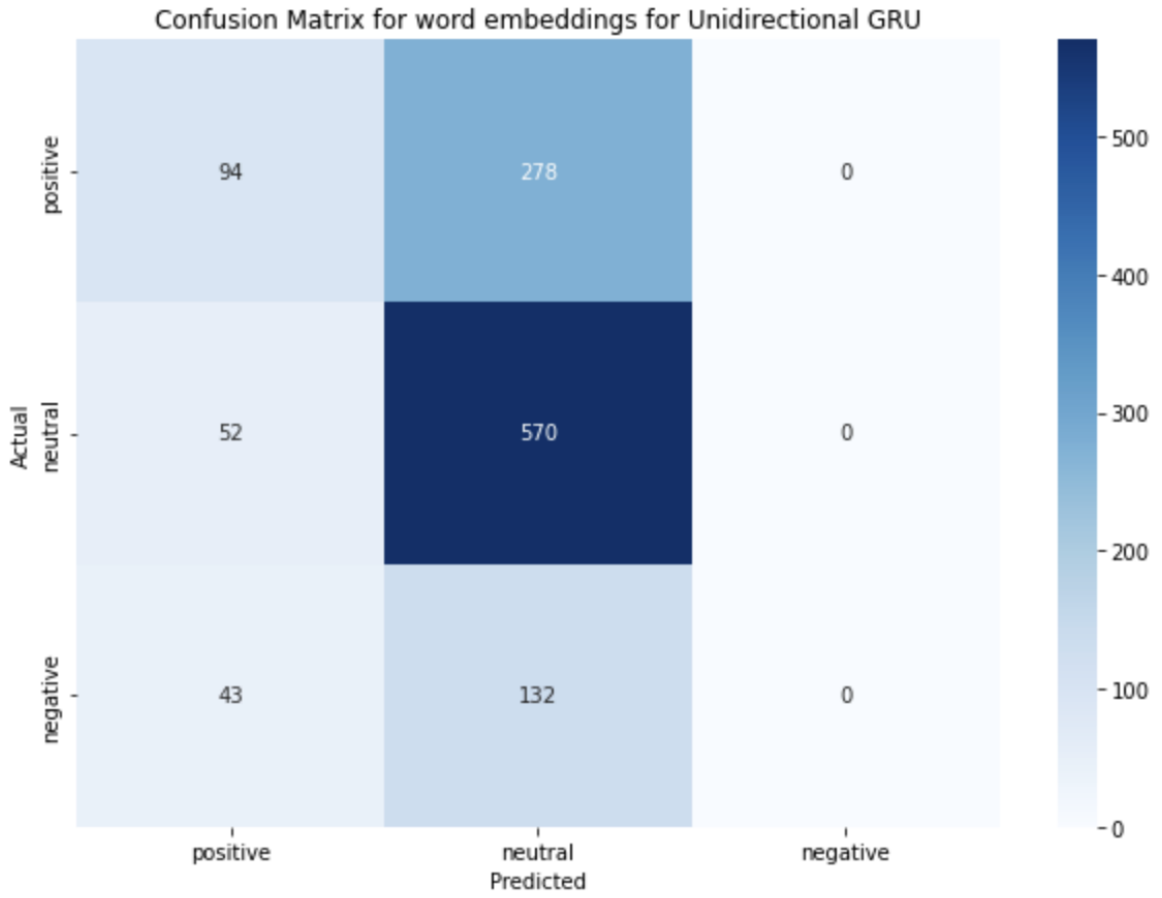
The test performances of PyTorch's Unidirectional GRU (GRU) and bidirectional GRU RNN implementations were compared based on sentiment classification accuracy. The unidirectional GRU model achieved an overall classification accuracy of approximately `76.22%`. It correctly classified positive sentiment reviews with an accuracy of 271 samples, negative sentiment reviews with an accuracy of 88 samples, and neutral sentiment reviews with an accuracy of 532 samples. In contrast, the bidirectional GRU model achieved a slightly higher overall classification accuracy of around `77.67%`. It demonstrated improved accuracy in classifying positive sentiment reviews, achieving an accuracy of 287 samples, negative sentiment reviews with an accuracy of 106 samples, and neutral sentiment reviews with an accuracy of 515 samples.

The Unidirectional GRU model processes the input sequence in a single direction, capturing sequential dependencies from past to future. On the other hand, the bidirectional GRU model processes the input sequence in both forward and backward directions, enabling it to capture dependencies from both past and future contexts. As a result, the bidirectional model exhibits better performance in capturing long-range dependencies and contextual information compared to the unidirectional model. However, this increased performance comes at the cost of higher computational complexity due to processing the input sequence in both directions.

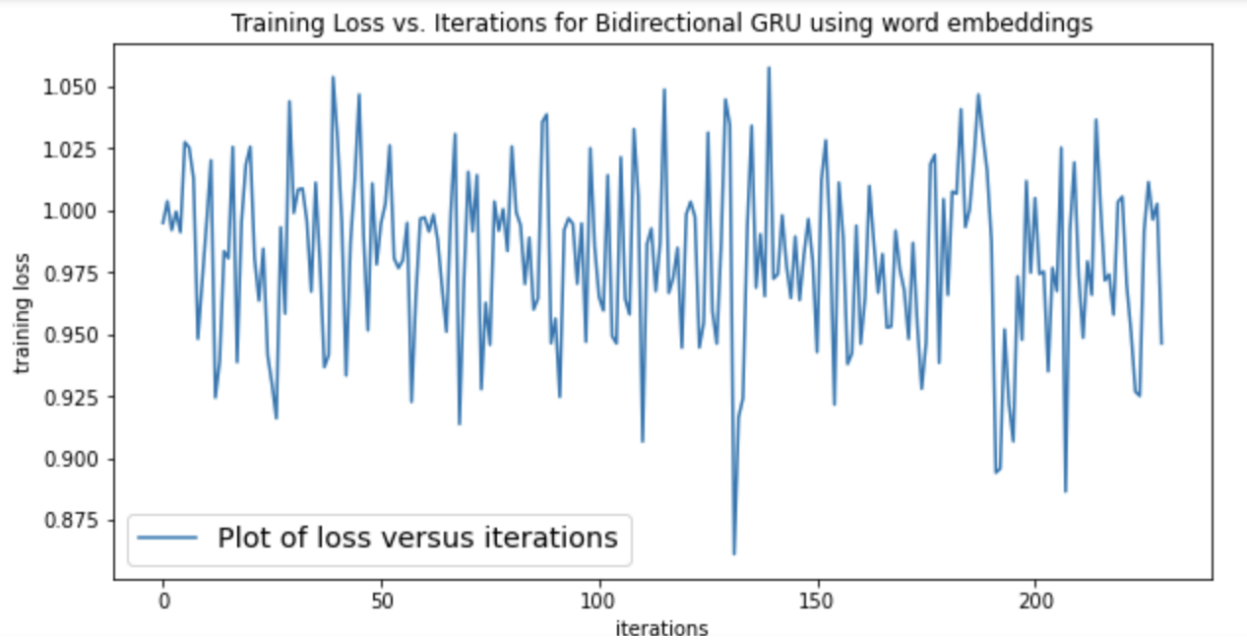
- Loss plot, classification accuracy and confusion matrix for Unidirectional GRU network using word embeddings.



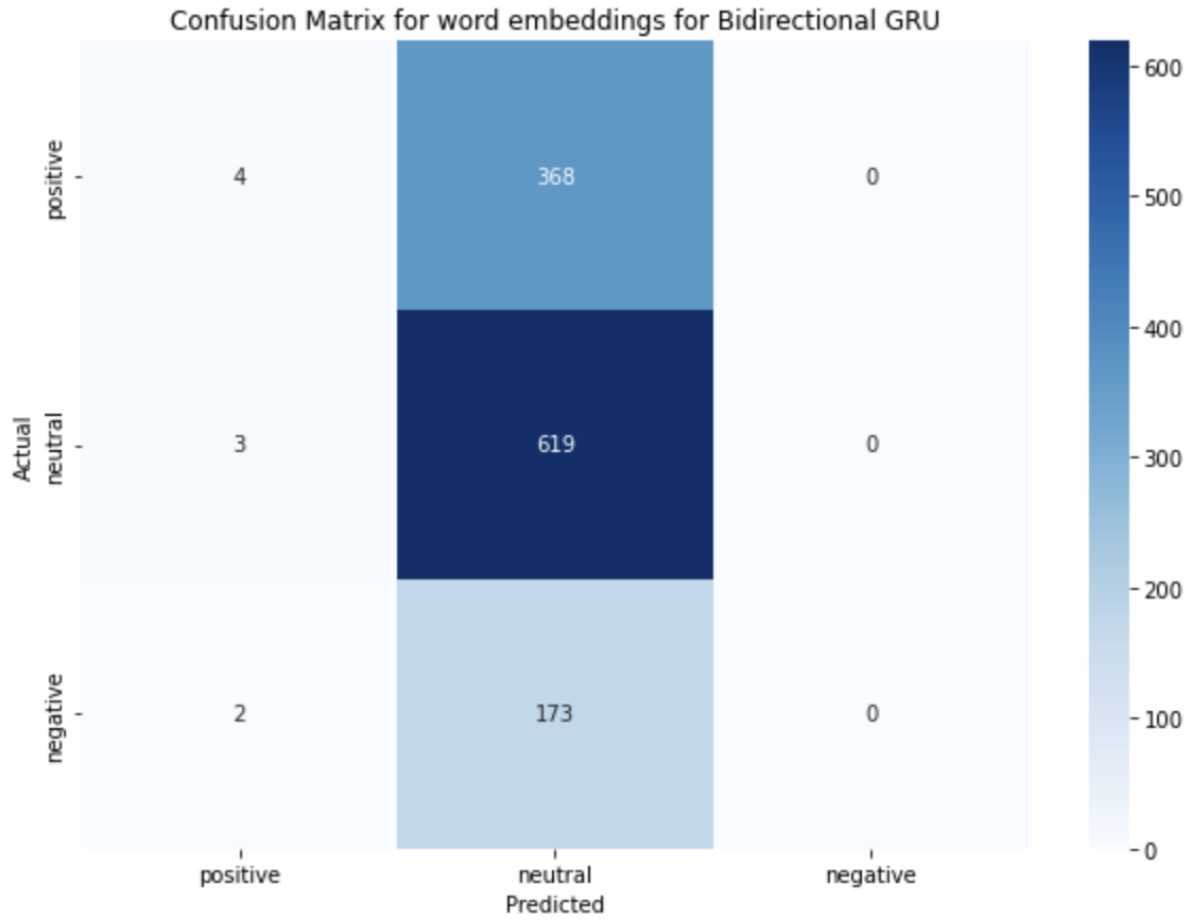

```
Overall classification accuracy: 56.80%
```



- Loss plot, classification accuracy and confusion matrix for Bidirectional GRU network using word embeddings.




```
Overall classification accuracy: 53.29%
```



As can be observed, when the Unidirectional GRU network using word embeddings is used, the accuracy achieved is `56.80%`, while when the Bidirectional GRU network using word embeddings is used, the accuracy achieved is `53.29%.` However, both models performed pretty well and provided similar kind of accuracy when trained over just 10 epochs. There can be a few possible reasons why the Bidirectional GRU did not perform as expected in this scenario. Class imbalance can be a possible reason that affected the performance of the bidirectional GRU network using word embeddings. In our case, where the dataset had a considerably higher number of neutral reviews and fewer positive and negative reviews, class imbalance could potentially predict the majority class (neutral reviews) more frequently, leading to lower accuracy on minority classes (positive and negative reviews). Class imbalance also skews the optimization

process during training, as the loss function may prioritize minimizing errors on the majority class. This can hinder the model's ability to learn meaningful representations for minority classes.

Overall, both models performed competitively, with the bidirectional GRU model showing a slight improvement in classifying sentiment reviews.

Using Amazon User Feedback Dataset:

Network and Training routine

The networks for the dataset sentiment_dataset_train_200 were run for 4 epochs, and the networks for the dataset sentiment_dataset_train_400 were run for 2 epochs, batch size of 1, input size of 300, hidden size of 100, output size of 2 (as we classified based on 2 review classes positive or negative), learning rate 1e-3, and trained used the Adam optimizer(betas = (0.5, 0.999)).

Unidirectional GRU Architecture used:

- **GRU Layer**: The GRU layer processes the input sequences and updates its hidden state iteratively. It takes the input tensor `x` and the initial hidden state `h` and produces an output tensor `out` and a final hidden state `h`. The input tensor `x` should have shape `(seq_len, batch_size, input_size)`.
- **Fully Connected Layer (Linear)**: The output of the GRU layer is passed through a fully connected (linear) layer (`nn.Linear`) to map the hidden state to the output space. The output size of this layer is specified by `output_size`, which typically corresponds to the number of classes in the classification task.
- **Activation Function (ReLU)**: The output of the linear layer is passed through the ReLU (Rectified Linear Unit) activation function (`nn.ReLU`) to introduce non-linearity to the model.
- **Output Layer (LogSoftmax)**: The output of the ReLU activation function is passed through a `LogSoftmax` layer (`nn.LogSoftmax`) to compute the log probabilities of each class. This layer normalizes the output into a probability distribution over the classes.

- **Initialization of Hidden State**: The `init_hidden` method initializes the hidden state of the GRU. It creates a tensor with dimensions `(num_layers, batch_size, hidden_size)` and initializes it with zeros.

```
#GRU with Embeddings
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
        self.gru = nn.GRU(input_size, hidden_size, num_layer
s)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

    def init_hidden(self):
        weight = next(self.parameters()).data
        #                  num_layers  batch_size    hidden_s
ize
        hidden = weight.new(  2,          1,         self.hid
den_size    ).zero_()
        return hidden
```

Bidirectional GRU Architecture used:

- **Bidirectional GRU Layer**: The GRU layer is configured to be bidirectional by setting `bidirectional=True` during initialization. This means that there are two sets of hidden states: one processing the input sequence from left to right, and the other processing it from right to left. The outputs of both directions are concatenated before being passed to the next layer.
- **Fully Connected Layer (Linear)**: The output of the bidirectional GRU layer is passed through a fully connected (linear) layer (`nn.Linear`) to map the concatenated hidden states to the output space. The output size of this layer is `2 * hidden_size`, where `2` represents the concatenation of the hidden states from both directions.
- **Activation Function (ReLU)**: The output of the linear layer is passed through the ReLU (Rectified Linear Unit) activation function (`nn.ReLU`) to introduce non-linearity to the model.
- **Output Layer (LogSoftmax)**: The output of the ReLU activation function is passed through a `LogSoftmax` layer (`nn.LogSoftmax`) to compute the log probabilities of each class. This layer normalizes the output into a probability distribution over the classes.
- **Initialization of Hidden State**: The `init_hidden` method initializes the hidden state of the bidirectional GRU. It creates a tensor with dimensions `(2*num_layers, batch_size, hidden_size)`, where `2*num_layers` represents the number of directions (2 for bidirectional) and layers. The tensor is initialized with zeros.

```
# BiGRU 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_layer
s, bidirectional=True)
        self.fc = nn.Linear(2*hidden_size, output_size)
```

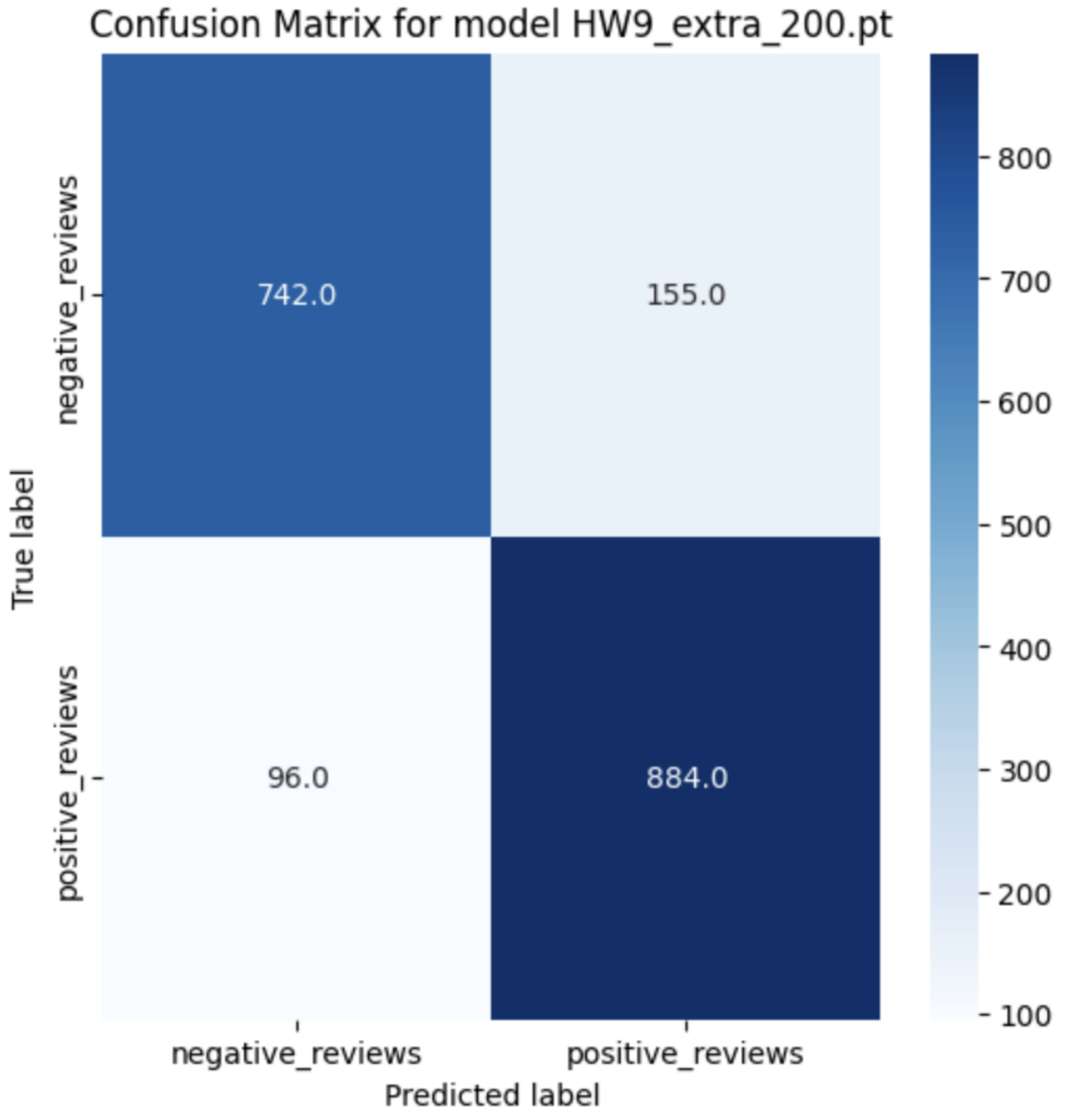
```
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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

Conclusion and Test Performance plots:

Loss plot, classification accuracy and confusion matrix for Unidirectional GRU network on sentiment_dataset_train_200 dataset:

Overall classification accuracy: 86.67%

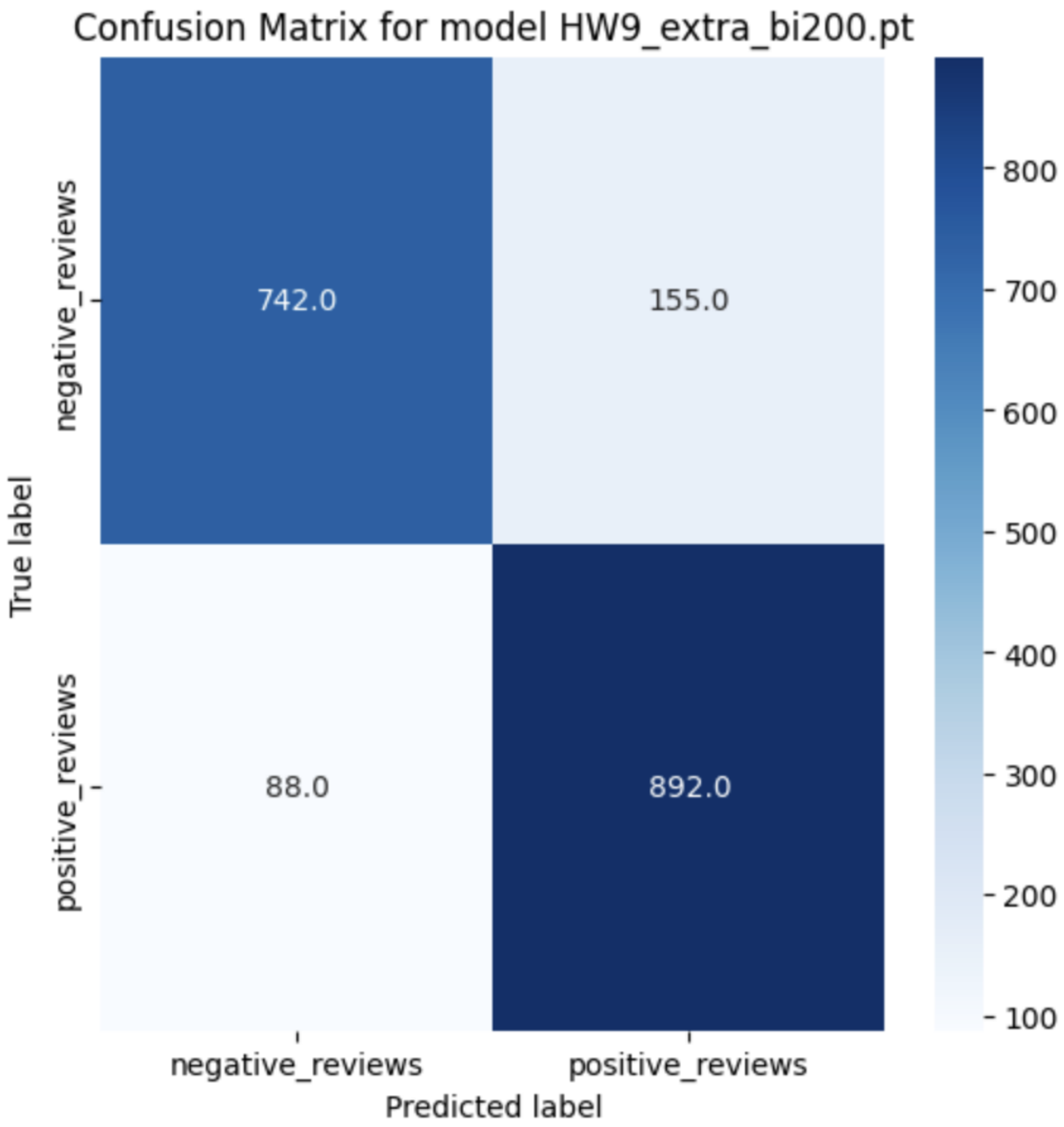
	predicted negative	predicted positive
true negative:	82.72%	17.28%
true positive:	9.796%	90.204%



Loss plot, classification accuracy and confusion matrix for Bidirectional GRU network on sentiment_dataset_train_200 dataset

Overall classification accuracy: 87.10%

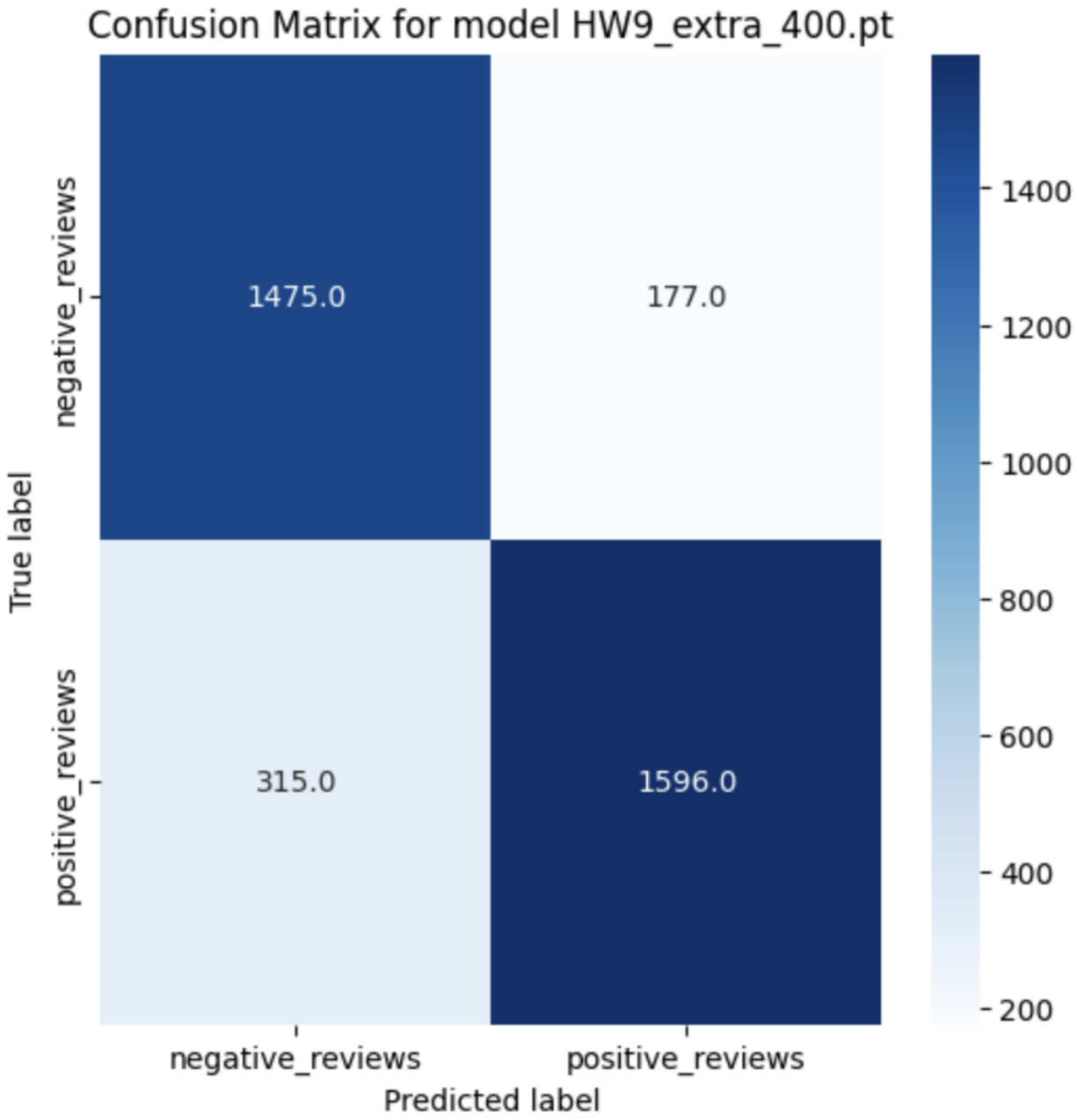
	predicted negative	predicted positive
true negative:	82.72%	17.28%
true positive:	8.98%	91.02%



Loss plot, classification accuracy and confusion matrix for Unidirectional GRU network on sentiment_dataset_train_400 dataset

Overall classification accuracy: 86.22%

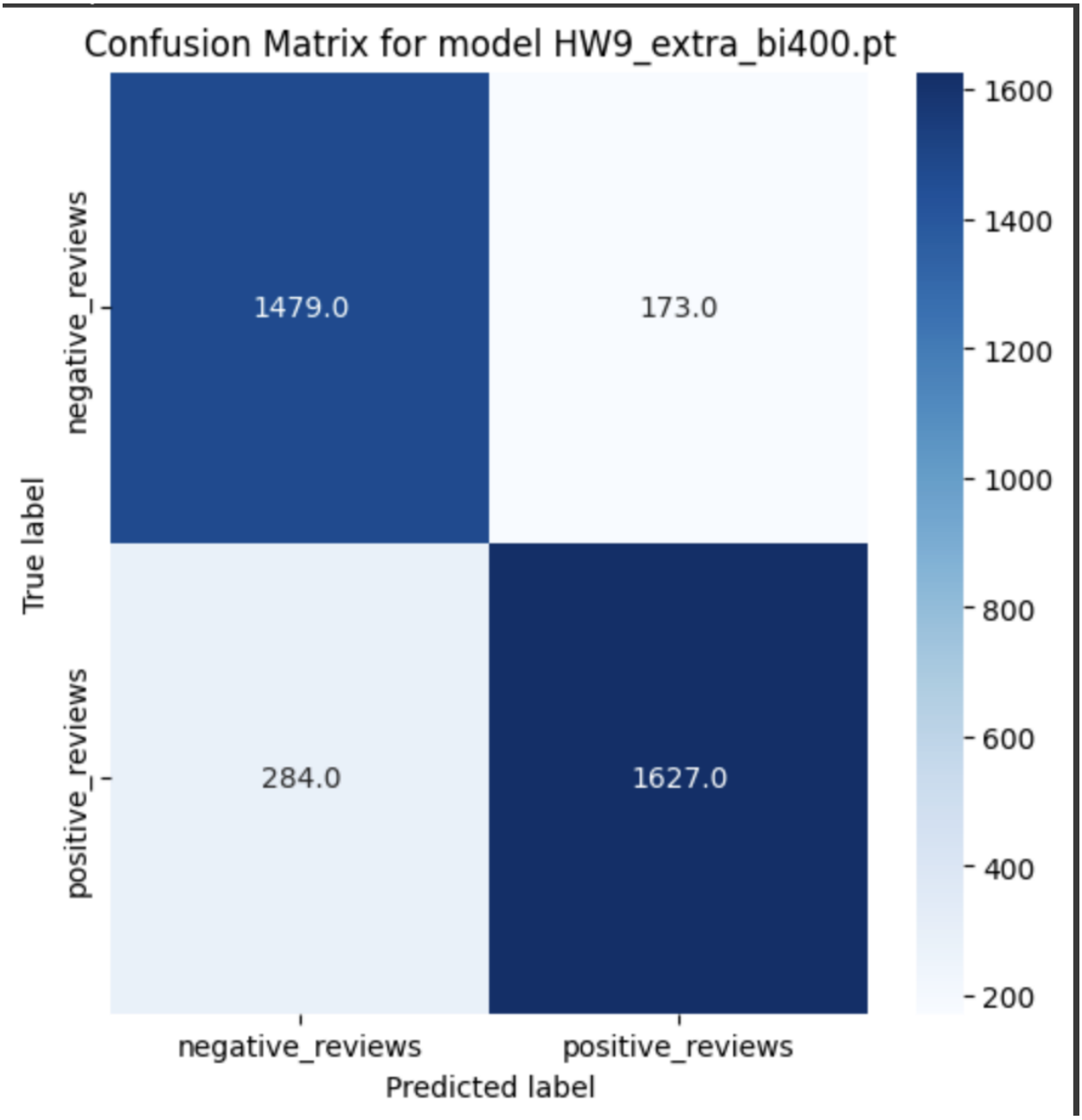
	predicted negative	predicted positive
true negative:	89.286%	10.714%
true positive:	16.484%	83.516%



Loss plot, classification accuracy and confusion matrix for Bidirectional GRU network on sentiment_dataset_train_400 dataset

Overall classification accuracy: 87.20%

	predicted negative	predicted positive
true negative:	89.528%	10.472%
true positive:	14.861%	85.139%



Comparison of the test performances of the both RNN implementations

Using a bidirectional scan made a difference in terms of test performance using the sentiment_dataset_train_200 and sentiment_dataset_train_400 dataset. As can be observed when the Unidirectional GRU network using

sentiment_dataset_train_200 dataset is tested, the accuracy achieved is `86.67%`, while when the Bidirectional GRU network using the sentiment_dataset_train_200 dataset is tested, the accuracy achieved is `87.10%`. Similarly, when the Unidirectional GRU network using sentiment_dataset_train_400 dataset is tested, the accuracy achieved is `86.22%`, while when the Bidirectional GRU network using the sentiment_dataset_train_400 dataset is tested, the accuracy achieved is `87.20%`. This means the accuracy is improved and when using Bidirectional Architecture. Also from the confusion matrix it can be observed that the classification is more clear between different classes and we can see better classified number of test samples in the Confusion matrix formed using the Bidirectional architecture.

Source Code:

```
get_ipython().system('pip install transformers')


#imports
import sys,os,os.path
import math
import random
import matplotlib.pyplot as plt
import time
import glob
import copy
import enum
import numpy as np
from PIL import Image
import torch.optim as optim
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import AdamW
import torchvision
```

```
from torch.utils.data import Dataset, DataLoader
import seaborn as sns
import pandas as pd
```

```
#The below code is used as was provided in the Homework 9 lab
manual handout.
import csv

sentences = []
sentiments = []
count = 0
with open('data.csv', 'r') as f:
    reader = csv.reader(f)
    # ignore the first line
    next(reader)
    for row in reader:
        sentences.append(row[0])
        sentiments.append(row[1])


print(sentences)


word_tokenized_sentences = [ sentence . split () for sentence
in sentences ]
print ( word_tokenized_sentences [:2])

max_len = max ([len ( sentence ) for sentence in
word_tokenized_sentences ])
padded_sentences = [ sentence + ['[PAD]'] * ( max_len - len (
sentence ) ) for sentence in
word_tokenized_sentences ]
print ( padded_sentences [:2])

from transformers import DistilBertTokenizer
```

```
model_ckpt = "distilbert-base-uncased"
distilbert_tokenizer = DistilBertTokenizer.from_pretrained(mo
del_ckpt)
bert_tokenized_sentences_ids = [ distilbert_tokenizer.encode
( sentence , padding ='max_length',truncation =True ,max_leng
th = max_len ) for sentence in sentences ]
print ( bert_tokenized_sentences_ids [:2])

bert_tokenized_sentences_tokens = [ distilbert_tokenizer.conv
ert_ids_to_tokens (sentence ) for sentence in bert_tokenized_
sentences_ids]
print ( bert_tokenized_sentences_tokens [:2])

vocab = {}
vocab['[PAD]'] = 0

print(vocab)

for sentence in padded_sentences:
    for token in sentence:
        if token not in vocab:
            vocab[token] = len(vocab)

# Convert the tokens to IDs
padded_sentences_ids = [[vocab[token] for token in sentence]
for sentence in padded_sentences]
print(padded_sentences_ids[:2])

from transformers import DistilBertModel
import torch

model_name = 'distilbert-base-uncased'
distilbert_model = DistilBertModel.from_pretrained(model_nam
e)
# Extract word embeddings
```

```
word_embeddings = []
# Convert padded sentence tokens into ids
for tokens in padded_sentences_ids:
    input_ids = torch.tensor(tokens).unsqueeze(0)
    with torch.no_grad():
        outputs = distilbert_model(input_ids)
    word_embeddings.append(outputs.last_hidden_state)

print(word_embeddings[0].shape)

from transformers import DistilBertModel
import torch

model_name = 'distilbert-base-uncased'
distilbert_model = DistilBertModel.from_pretrained(model_nam
e)
# Subword embeddings extraction
subword_embeddings = []
for tokens in bert_tokenized_sentences_ids:
    input_ids = torch.tensor(tokens).unsqueeze(0)
    with torch.no_grad():
        outputs = distilbert_model(input_ids)
    subword_embeddings.append(outputs.last_hidden_state)

print(subword_embeddings[0].shape)
```

```
# Encode sentiments as one-hot vectors
def encode_sentiment(sentiment):
    if sentiment == 'positive':
        return np.array([1, 0, 0])
    elif sentiment == 'neutral':
        return np.array([0, 1, 0])
    elif sentiment == 'negative':
        return np.array([0, 0, 1])
    else:
```

```
        raise ValueError("Invalid sentiment value")


sentiments_one_hot =torch.tensor( [encode_sentiment(sentimen
t) for sentiment in sentiments])



# Define custom Sentiment Analysis Dataset class
class CustomSentimentAnalysisDataset(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):
        word_embedding =(self.word_embeddings[idx])
        sentiment =(self.sentiments[idx])

        return word_embedding, sentiment



# Split the data into training and testing sets for word embe
ddings
from sklearn.model_selection import train_test_split
train_word_embeddings, test_word_embeddings, train_sentiment
s, test_sentiments = train_test_split(word_embeddings, sentim
ents_one_hot, test_size=0.2, random_state=42)

# Create DataLoader for training set for word embeddings
train_word_dataset = CustomSentimentAnalysisDataset(train_wor
d_embeddings, train_sentiments)
train_word_dataloader = DataLoader(train_word_dataset, batch_
```

```
size=1, shuffle=True)

# Create DataLoader for testing set for word embeddings
test_word_dataset = CustomSentimentAnalysisDataset(test_word_
embeddings, test_sentiments)
test_word_dataloader = DataLoader(test_word_dataset, batch_si
ze=1, shuffle=True)



# Split the data into training and testing sets for subword e
mbeddings
from sklearn.model_selection import train_test_split
train_subword_embeddings, test_subword_embeddings, train_sent
iments, test_sentiments = train_test_split(subword_embedding
s, sentiments_one_hot, test_size=0.2, random_state=42)

# Create DataLoader for training set for subword embeddings
train_subword_dataset = CustomSentimentAnalysisDataset(train_
subword_embeddings, train_sentiments)
train_subword_dataloader = DataLoader(train_subword_dataset,
batch_size=1, shuffle=True)

# Create DataLoader for testing set for subword embeddings
test_subword_dataset = CustomSentimentAnalysisDataset(test_su
bword_embeddings, test_sentiments)
test_subword_dataloader = DataLoader(test_subword_dataset, ba
tch_size=1, shuffle=True)
```

```
# Code for GRU network which is inspired from Dr. Avinash Ka
k's DL Studio Module with slight modifications
class GRUnet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size,
num_layers, drop_prob=0.2):
```

```
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layer
s)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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




# Code for GRU training routine which is inspired from Dr. Av
inash Kak's DL Studio Module with slight modifications
def run_code_for_training_with_GRU( net,train_dataloader,devi
ce,model_name, display_train_loss=True):
    filename_for_out = "performance_numbers_GRU_" + str(10) +
".txt"
    FILE = open(filename_for_out, 'w')
    net.to(device)

    criterion = nn.NLLLoss()
    accum_times = []
```

```
    optimizer = optim.Adam(net.parameters(), lr=1e-4, betas =
(0.8, 0.999))
    start_time = time.perf_counter()
    training_loss_tally = []
    for epoch in range(10):
        print("")
        running_loss = 0.0
        for i, data in enumerate(train_dataloader):
            hidden = net.init_hidden().to(device)

            review_tensor,sentiment = data
            review_tensor = review_tensor[0].to(device)
            sentiment = sentiment[0].to(device)

            optimizer.zero_grad()
            for k in range(review_tensor.shape[1]):

                output, hidden = net(torch.unsqueeze(torch.un
squeeze(review_tensor[0,k],0),0).to(device), hidden)
            loss = criterion(output, torch.argmax(sentiment.u
nsqueeze(0),1))
            running_loss += loss.item()
            loss.backward(retain_graph=True)
            optimizer.step()

            if i % 200 == 199:
                avg_loss = running_loss / float(200)
                training_loss_tally.append(avg_loss)
                current_time = time.perf_counter()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time: %4d
secs]     loss: %.5f" % (epoch+1,i+1, time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                FILE.write("%.3f\n" % avg_loss)
                FILE.flush()
                running_loss = 0.0
```

```
    print("Total Training Time: {}".format(str(sum(accum_time
s))))
    print("\nFinished Training\n")
    torch.save(net.state_dict(), f"{model_name}.pt")
    if display_train_loss:
        plt.figure(figsize=(10,5))
        plt.title("Training Loss vs. Iterations for unidirect
ional GRU using word embeddings")
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")

        plt.legend(["Plot of loss versus iterations"], fontsi
ze="x-large")
        plt.savefig("training_loss.png")
        plt.show()
    return training_loss_tally
```

```
# Code for GRU testing routine which is inspired from Dr. Avi
nash Kak's DL Studio Module with slight modifications
def run_code_for_testing_text_classification_with_GRU(net, te
st_dataloader,model_name, device):
    net.load_state_dict(torch.load(f"{model_name}.pt"))
    net.to(device)
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    neutral_total = 0
    confusion_matrix = torch.zeros(3, 3)

    with torch.no_grad():
        for i, data in enumerate(test_dataloader):
            review_tensor, sentiment = data
            review_tensor = review_tensor[0].to(device)
            sentiment = sentiment[0].to(device)
```

```
            hidden = net.init_hidden().to(device)

            for k in range(review_tensor.shape[1]):
                output, hidden = net(torch.unsqueeze(torch.un
squeeze(review_tensor[0,k],0),0), hidden)

            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if i % 100 == 99:
                print("   [i=%d]    predicted_label=%d
gt_label=%d\n\n" % (i+1, predicted_idx, gt_idx))

            if predicted_idx == gt_idx:
                classification_accuracy += 1

            if gt_idx == 0:
                positive_total += 1
            elif gt_idx == 1:
                neutral_total += 1
            elif gt_idx == 2:
                negative_total += 1

            confusion_matrix[gt_idx, predicted_idx] += 1

    classification_accuracy /= len(test_dataloader)
    print("\nOverall classification accuracy: %0.2f%%" % (cla
ssification_accuracy * 100))

    out_percent = np.zeros((3,3), dtype='float')
    out_percent[0,:] = 100 * confusion_matrix[0,:] / negative
_total
    out_percent[1,:] = 100 * confusion_matrix[1,:] / neutral_
total
    out_percent[2,:] = 100 * confusion_matrix[2,:] / positive
_total
```

```
    print("\n\nNumber of negative reviews tested: %d" % negat
ive_total)
    print("\nNumber of neutral reviews tested: %d" % neutral_
total)
    print("\nNumber of positive reviews tested: %d" % positiv
e_total)

    print("\n\nDisplaying the confusion matrix:\n")


    plt.figure(figsize=(10, 7))
    sns.heatmap(confusion_matrix, annot=True, fmt='g', cmap
='Blues', xticklabels=['positive', 'neutral', 'negative'],
                yticklabels=['positive', 'neutral', 'negativ
e'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix for word embeddings for Unidi
rectional GRU')#edit the name of corresponding model for titl
e label
    plt.show()


# Initialize the Unidirectional GRU model
model = GRUnet(768, hidden_size=100, output_size=3, num_layer
s=3)
device=torch.device("cuda:0" if (torch.cuda.is_available()) e
lse "cpu")
number_of_learnable_params = sum(p.numel() for p in model.par
ameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layer
s)
print("\nThe number of learnable parameters in the model: %d"
```

```
% number_of_learnable_params)



## TRAINING for unidirectional GRU for subword embeddings:
avg_loss_uni_subword= run_code_for_training_with_GRU(net=mode
l,train_dataloader=train_subword_dataloader, device=device, m
odel_name ="unigru_subword", display_train_loss=True)

## Testing for unidirectional GRU for subword embeddings:
run_code_for_testing_text_classification_with_GRU(net=model,t
est_dataloader=test_subword_dataloader, model_name="unigru_su
bword", device=device)




## TRAINING for unidirectional GRU for word embeddings:
avg_loss_uni_word= run_code_for_training_with_GRU(net=model,t
rain_dataloader=train_word_dataloader, device=device, model_n
ame ="unigru_word", display_train_loss=True)

## TESTING for unidirectional GRU for word embeddings:
run_code_for_testing_text_classification_with_GRU(net=model,t
est_dataloader=test_word_dataloader, model_name="unigru_wor
d", device=device)
```

```
# Code for Bidirectional GRU Network which is inspired from D
r. Avinash Kak's DL Studio Module with slight modifications
# to the Unidirectional GRU
class BiGRUnet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size,
num_layers, drop_prob=0.2):
```

```
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = True

        self.gru = nn.GRU(input_size, hidden_size, num_layer
s, dropout=drop_prob, bidirectional=True) # True for bidirect
ional GRU
        self.fc = nn.Linear(hidden_size * 2, output_size)  #
Multiply by 2 for bidirectional
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = torch.cat((out[:, -1, :self.hidden_size], out
[:, 0, self.hidden_size:]), dim=1)
        out = self.fc(self.relu(out))
        out = self.logsoftmax(out)
        return out, h

    def init_hidden(self, batch_size=1):
        weight = next(self.parameters()).data
        num_directions = 2 if self.bidirectional else 1
        hidden = weight.new(self.num_layers * num_directions,
1, self.hidden_size).zero_()
        return hidden




# Code for Bidirectional GRU training routine which is inspir
ed from Dr. Avinash Kak's DL Studio Module with slight modifi
cations
def run_code_for_training_with_BiGRU( net,train_dataloader,de
vice,model_name, display_train_loss=True):
```

```
    filename_for_out = "performance_numbers_BiGRU" + str(10)
+ ".txt"
    FILE = open(filename_for_out, 'w')
    net.to(device)
    criterion = nn.NLLLoss()
    accum_times = []
    optimizer = optim.Adam(net.parameters(), lr=1e-4, betas =
(0.8, 0.999))
    start_time = time.perf_counter()
    training_loss_tally = []
    for epoch in range(10):
        print("")
        running_loss = 0.0
        for i, data in enumerate(train_dataloader):
            hidden = net.init_hidden().to(device)

            review_tensor,sentiment = data
            review_tensor = review_tensor[0].to(device)
            sentiment = sentiment[0].to(device)

            optimizer.zero_grad()
            for k in range(review_tensor.shape[1]):
                output, hidden = net(torch.unsqueeze(torch.un
squeeze(review_tensor[0,k],0),0).to(device), hidden)
            loss = criterion(output, torch.argmax(sentiment.u
nsqueeze(0),1))
            running_loss += loss.item()
            loss.backward(retain_graph=True)
            optimizer.step()

            if i % 200 == 199:
                avg_loss = running_loss / float(200)
                training_loss_tally.append(avg_loss)
                current_time = time.perf_counter()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time: %4d
```

```
secs]     loss: %.5f" % (epoch+1,i+1, time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                FILE.write("%.3f\n" % avg_loss)
                FILE.flush()
                running_loss = 0.0
    print("Total Training Time: {}".format(str(sum(accum_time
s))))
    print("\nFinished Training\n")
    torch.save(net.state_dict(), f"{model_name}.pt")
    if display_train_loss:
        plt.figure(figsize=(10,5))
        plt.title("Training Loss vs. Iterations for Bidirecti
onal GRU using word embeddings") #edit the name of correspond
ing model for title label
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")
        plt.legend(["Plot of loss versus iterations"], fontsi
ze="x-large")
        plt.savefig("training_loss.png")
        plt.show()
    return training_loss_tally
```

```
# Code for Bidirectional GRU testing routine which is inspire
d from Dr. Avinash Kak's DL Studio Module with slight modific
ations
def run_code_for_testing_text_classification_with_BiGRU(net,
test_dataloader,model_name, device):
    net.load_state_dict(torch.load(f"{model_name}.pt"))
    net.to(device)
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    neutral_total = 0
```

```
    confusion_matrix = torch.zeros(3, 3)

    with torch.no_grad():
        for i, data in enumerate(test_dataloader):
            review_tensor, sentiment = data
            review_tensor = review_tensor[0].to(device)
            sentiment = sentiment[0].to(device)
            hidden = net.init_hidden().to(device)

            for k in range(review_tensor.shape[1]):
                output, hidden = net(torch.unsqueeze(torch.un
squeeze(review_tensor[0,k],0),0), hidden)

            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if i % 100 == 99:
                print("   [i=%d]    predicted_label=%d
gt_label=%d\n\n" % (i+1, predicted_idx, gt_idx))

            if predicted_idx == gt_idx:
                classification_accuracy += 1

            if gt_idx == 0:
                positive_total += 1
            elif gt_idx == 1:
                neutral_total += 1
            elif gt_idx == 2:
                negative_total += 1

            confusion_matrix[gt_idx, predicted_idx] += 1

    classification_accuracy /= len(test_dataloader)
    print("\nOverall classification accuracy: %0.2f%%" % (cla
ssification_accuracy * 100))

    out_percent = np.zeros((3,3), dtype='float')
```

```
    out_percent[0,:] = 100 * confusion_matrix[0,:] / negative
_total
    out_percent[1,:] = 100 * confusion_matrix[1,:] / neutral_
total
    out_percent[2,:] = 100 * confusion_matrix[2,:] / positive
_total

    print("\n\nNumber of negative reviews tested: %d" % negat
ive_total)
    print("\nNumber of neutral reviews tested: %d" % neutral_
total)
    print("\nNumber of positive reviews tested: %d" % positiv
e_total)

    print("\n\nDisplaying the confusion matrix:\n")



    plt.figure(figsize=(10, 7))
    sns.heatmap(confusion_matrix, annot=True, fmt='g', cmap
='Blues', xticklabels=['positive', 'neutral', 'negative'],
                yticklabels=['positive', 'neutral', 'negativ
e'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix for subword embeddings for Bi
directional GRU')#edit the name of corresponding model for ti
tle label
    plt.show()




# Initialize the Bidirectional BiGRU model
model1 = BiGRUnet(768, hidden_size=100, output_size=3, num_la
yers=3)
device=torch.device("cuda:0" if (torch.cuda.is_available()) e
```

```
lse "cpu")
number_of_learnable_params = sum(p.numel() for p in model1.pa
rameters() if p.requires_grad)

num_layers = len(list(model1.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layer
s)
print("\nThe number of learnable parameters in the model: %d"
% number_of_learnable_params)




## TRAINING for bidirectional GRU for subword embeddings:
avg_loss_bidirec_subword= run_code_for_training_with_BiGRU(ne
t=model1,train_dataloader=train_subword_dataloader, device=de
vice, model_name ="bigru_subword", display_train_loss=True)

## TESTING for bidirectional GRU for subword embeddings:
run_code_for_testing_text_classification_with_BiGRU(net=model
1,test_dataloader=test_subword_dataloader, model_name="bigru_
subword", device=device)




## TRAINING for bidirectional GRU for word embeddings:
avg_loss_bidirec_word= run_code_for_training_with_BiGRU(net=m
odel1,train_dataloader=train_word_dataloader, device=device,
model_name ="bigru_word", display_train_loss=True)


## TESTING for bidirectional GRU for word embeddings:
run_code_for_testing_text_classification_with_BiGRU(net=model
```

```
1,test_dataloader=test_word_dataloader, model_name="bigru_wor
d", device=device)
```

Source Code for Extra Credit:

```
#imports
import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt
import sys
import torch.nn as nn
import torch.optim as optim
import copy
import time
from tqdm import tqdm
import gzip
import pickle
import gensim.downloader as gen_api
import gensim.downloader as genapi
from gensim.models import KeyedVectors
import seaborn as sns


seed = 10
random.seed(seed)
np.random.seed(seed)
```

```
# Code for dataset class which is inspired from Dr. Avinash K
ak's DL Studio Module with slight modifications.
class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, root, dataset_file, mode = 'train', pa
```

```
th_to_saved_embeddings=None):
        super(SentimentAnalysisDataset, self).__init__()
        self.path_to_saved_embeddings = path_to_saved_embeddi
ngs
        self.mode = mode
        root_dir = root
        f = gzip.open(root_dir + dataset_file, 'rb')
        dataset = f.read()
        if path_to_saved_embeddings is not None:
            if os.path.exists(path_to_saved_embeddings + 'vec
tors.kv'):
                self.word_vectors = KeyedVectors.load(path_to
_saved_embeddings + 'vectors.kv')
            else:
                self.word_vectors = genapi.load("word2vec-goo
gle-news-300")

                self.word_vectors.save(path_to_saved_embeddin
gs + 'vectors.kv')
        if mode == 'train':
            if sys.version_info[0] == 3:
                self.positive_reviews_train, self.negative_re
views_train, self.vocab = pickle.loads(dataset, encoding='lat
in1')
            else:
                self.positive_reviews_train, self.negative_re
views_train, self.vocab = pickle.loads(dataset)
            self.categories = sorted(list(self.positive_revie
ws_train.keys()))
            self.category_sizes_train_pos = {category : len(s
elf.positive_reviews_train[category]) for category in self.ca
tegories}
            self.category_sizes_train_neg = {category : len(s
elf.negative_reviews_train[category]) for category in self.ca
tegories}
            self.indexed_dataset_train = []
```

```
            for category in self.positive_reviews_train:
                for review in self.positive_reviews_train[cat
egory]:
                    self.indexed_dataset_train.append([revie
w, category, 1])
            for category in self.negative_reviews_train:
                for review in self.negative_reviews_train[cat
egory]:
                    self.indexed_dataset_train.append([revie
w, category, 0])
            random.shuffle(self.indexed_dataset_train)
        elif mode == 'test':
            if sys.version_info[0] == 3:
                self.positive_reviews_test, self.negative_rev
iews_test, self.vocab = pickle.loads(dataset, encoding='latin
1')
            else:
                self.positive_reviews_test, self.negative_rev
iews_test, self.vocab = pickle.loads(dataset)
            self.vocab = sorted(self.vocab)
            self.categories = sorted(list(self.positive_revie
ws_test.keys()))
            self.category_sizes_test_pos = {category : len(se
lf.positive_reviews_test[category]) for category in self.cate
gories}
            self.category_sizes_test_neg = {category : len(se
lf.negative_reviews_test[category]) for category in self.cate
gories}
            self.indexed_dataset_test = []
            for category in self.positive_reviews_test:
                for review in self.positive_reviews_test[cate
gory]:
                    self.indexed_dataset_test.append([review,
category, 1])
            for category in self.negative_reviews_test:
                for review in self.negative_reviews_test[cate
```

```
gory]:
                    self.indexed_dataset_test.append([review,
category, 0])
            random.shuffle(self.indexed_dataset_test)

    def review_to_tensor(self, review):
        list_of_embeddings = []
        for i,word in enumerate(review):
            if word in self.word_vectors.key_to_index:
                embedding = self.word_vectors[word]
                list_of_embeddings.append(np.array(embeddin
g))
            else:
                next
        review_tensor = torch.FloatTensor(list_of_embeddings)
        return review_tensor

    def sentiment_to_tensor(self, sentiment):
        sentiment_tensor = torch.zeros(2)
        if sentiment == 1:
            sentiment_tensor[1] = 1
        elif sentiment == 0:
            sentiment_tensor[0] = 1
        sentiment_tensor = sentiment_tensor.type(torch.long)
        return sentiment_tensor

    def __len__(self):
        if self.mode == 'train':
            return len(self.indexed_dataset_train)
        elif self.mode == 'test':
            return len(self.indexed_dataset_test)

    def __getitem__(self, idx):
        sample = self.indexed_dataset_train[idx] if self.mode
== 'train' else self.indexed_dataset_test[idx]
        review = sample[0]
```

```
        review_category = sample[1]
        review_sentiment = sample[2]
        review_sentiment = self.sentiment_to_tensor(review_se
ntiment)
        review_tensor = self.review_to_tensor(review)
        category_index = self.categories.index(review_categor
y)
        sample = {'review'       : review_tensor,
                  'category'     : category_index, # should
be converted to tensor, but not yet used
                  'sentiment'    : review_sentiment }
        return sample
```

```
# Code for torch.nn GRU with Embeddings inspired from Dr. Avi
nash Kak's DL Studio Module.
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
        self.gru = nn.GRU(input_size, hidden_size, num_layer
s)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

    def init_hidden(self):
```

```
        weight = next(self.parameters()).data
        #                  num_layers  batch_size    hidden_s
ize
        hidden = weight.new(  2,          1,         self.hid
den_size    ).zero_()
        return hidden

# Code for Bidirectional GRU Network with Embeddings which is
inspired from Dr. Avinash Kak's DL Studio Module
# with slight modifications to the Unidirectional GRU
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_layer
s, bidirectional=True)
        self.fc = nn.Linear(2*hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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

```
#Code for GRU with embeddings training routine which is inspi
red from Dr. Avinash Kak's DL Studio Module with slight modif
ications.
def run_code_for_training_for_text_classification_with_GRU(de
vice, net, dataloader, model_name, epochs, display_interval):
    net = net.to(device)
    criterion = nn.NLLLoss()
    accum_times = []
    optimizer = optim.Adam(net.parameters(), lr=1e-3, betas =
(0.5, 0.999))
    training_loss_tally = []
    start_time = time.perf_counter()
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(dataloader):
            review_tensor,category,sentiment = data['revie
w'], data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            optimizer.zero_grad()
            hidden = net.init_hidden().to(device)
            output, hidden = net(torch.unsqueeze(review_tenso
r[0], 1), hidden)
            loss = criterion(output, torch.argmax(sentiment,
1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()

            if (i+1) % display_interval == 0:
                avg_loss = running_loss / float(display_inter
val)
                training_loss_tally.append(avg_loss)
                current_time = time.perf_counter()
```

```
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time:%4d
secs] loss: %.5f" % (epoch+1,i+1, time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                running_loss = 0.0

    torch.save(net.state_dict(), os.path.join('/content/driv
e/My Drive/saved_models',f'{model_name}.pt'))

    print("Total Training Time: {}".format(str(sum(accum_time
s))))
    print("\nFinished Training\n\n")
    plt.figure(figsize=(10,5))
    plt.title(f"Training Loss vs. Iterations - {model_name}")
    plt.plot(training_loss_tally)
    plt.xlabel("Iterations")
    plt.ylabel("Training loss")
    plt.legend()
    plt.savefig(f"/content/drive/My Drive/training_loss_{mode
l_name}.png")
    plt.show()

    return training_loss_tally
```

```
#Code for GRU with embeddings testing routine which is inspir
ed from Dr. Avinash Kak's DL Studio Module with slight modifi
cations.
def run_code_for_testing_text_classification_with_GRU(device,
net, model_path, dataloader, model_name, display_interval):
    net.load_state_dict(torch.load(model_path))
    net = net.to(device)
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    confusion_matrix = torch.zeros(2,2)
```

```
    with torch.no_grad():
        for i, data in enumerate(dataloader):
            review_tensor,category,sentiment = data['revie
w'], data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            hidden = net.init_hidden().to(device)
            output, hidden = net(torch.unsqueeze(review_tenso
r[0], 1), hidden)
            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if (i+1) % display_interval == 0:
                print("   [i=%d]    predicted_label=%d
gt_label=%d" % (i+1, predicted_idx,gt_idx))
            if predicted_idx == gt_idx:
                classification_accuracy += 1
            if gt_idx == 0:
                negative_total += 1
            elif gt_idx == 1:
                positive_total += 1
            confusion_matrix[gt_idx,predicted_idx] += 1
    print("\nOverall classification accuracy: %0.2f%%" %  (fl
oat(classification_accuracy) * 100 /float(i)))
    out_percent = np.zeros((2,2), dtype='float')
    out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0]
/ float(negative_total))
    out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1]
/ float(negative_total))
    out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0]
/ float(positive_total))
    out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1]
/ float(positive_total))
    print("\n\nNumber of positive reviews tested: %d" % posit
ive_total)
    print("\n\nNumber of negative reviews tested: %d" % negat
```

```
ive_total)
    print("\n\nDisplaying the confusion matrix:\n")
    out_str = "                      "
    out_str +=  "%18s    %18s" % ('predicted negative', 'pred
icted positive')
    print(out_str + "\n")
    for i,label in enumerate(['true negative', 'true positiv
e']):
        out_str = "%12s:  " % label
        for j in range(2):
            out_str +=  "%18s%%" % out_percent[i,j]
        print(out_str)

    labels = []
    classes=['negative_reviews', 'positive_reviews']
    num_classes = len(classes)
    for row in range(num_classes):
        rows = []
        total_labels =  np.sum(confusion_matrix.numpy()[row])
        for col in range(num_classes):
            count = confusion_matrix.numpy()[row][col]
            label = str(count)
            rows.append(label)
        labels.append(rows)
    labels = np.asarray(labels)

    plt.figure(figsize=(6, 6))
    sns.heatmap(confusion_matrix.numpy(), annot=labels, fmt
="", cmap="Blues", cbar=True,
                xticklabels=classes, yticklabels=classes)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title(f'Confusion Matrix for model {model_name}')
    plt.show()

# training dataset for sentiment_dataset_train_200
```

```
train_dataset_200 = SentimentAnalysisDataset('/content/drive/
My Drive/','sentiment_dataset_train_200.tar.gz',
                                             path_to_saved_embedd
ings = '/content/drive/My Drive/word2vec/')



# testing dataset for sentiment_dataset_train_200
test_dataset_200 = SentimentAnalysisDataset('/content/drive/M
y Drive/', 'sentiment_dataset_test_200.tar.gz',
                                           mode = 'test', path_to
_saved_embeddings = '/content/drive/My Drive/word2vec/')



# Create custom training/testing dataloader for sentiment_dat
aset_train_200
train_data_loader = torch.utils.data.DataLoader(train_dataset
_200, batch_size=1, shuffle=True, num_workers=1)
test_data_loader = torch.utils.data.DataLoader(test_dataset_2
00, batch_size=1, shuffle=True, num_workers=1)

# Initialize GRU with Embeddings
device = torch.device('cuda') if torch.cuda.is_available() el
se torch.device('cpu')
print(f"Device: {device}")


model = GRUnetWithEmbeddings(input_size=300, hidden_size=100,
output_size=2, num_layers=2)

epochs = 4
display_interval = 500

# Count the number of learnable parameters and layers
number_of_learnable_params = sum(p.numel() for p in model.par
```

```
ameters() if p.requires_grad)
print("\nThe number of learnable parameters in the model:", n
umber_of_learnable_params)


num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model:", num_layers)

# Training for sentiment_dataset_train_200
net1_losses = run_code_for_training_for_text_classification_w
ith_GRU(device, model, dataloader=train_data_loader,
                                        model_name='HW9_extra
_200', epochs=epochs, display_interval=display_interval)

# Testing for sentiment_dataset_train_200
save_path = '/content/drive/My Drive/saved_models/HW9_extra_2
00.pt'
run_code_for_testing_text_classification_with_GRU(device, mod
el, dataloader=test_data_loader,
                               display_interval=display_interva
l, model_path=save_path, model_name='HW9_extra_200.pt')

# Initialize BiGRU with Embeddings
device = torch.device('cuda') if torch.cuda.is_available() el
se torch.device('cpu')
print(f"Device: {device}")

model = BiGRUnetWithEmbeddings(input_size=300, hidden_size=10
0, output_size=2, num_layers=2)

epochs = 4
display_interval = 500

# Count the number of learnable parameters
number_of_learnable_params = sum(p.numel() for p in model.par
ameters() if p.requires_grad)
```

```
print("\nThe number of learnable parameters in the model:", n
umber_of_learnable_params)

# Count the number of layers
num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model:", num_layers)

# Training for sentiment_dataset_train_200
net1_losses = run_code_for_training_for_text_classification_w
ith_GRU(device, model, dataloader=train_data_loader,
                                        model_name='HW9_extra
_bi200', epochs=epochs, display_interval=display_interval)

# Testing for sentiment_dataset_train_200
save_path = '/content/drive/My Drive/saved_models/HW9_extra_b
i200.pt'
run_code_for_testing_text_classification_with_GRU(device, mod
el, dataloader=test_data_loader,
                               display_interval=display_interva
l, model_path=save_path, model_name='HW9_extra_bi200.pt')
```

```
# training dataset for sentiment_dataset_train_400
train_dataset_400 = SentimentAnalysisDataset('/content/drive/
My Drive/','sentiment_dataset_train_400.tar.gz',
                                          path_to_saved_embedd
ings = '/content/drive/My Drive/word2vec/')



# testing dataset for sentiment_dataset_train_400
test_dataset_400 = SentimentAnalysisDataset('/content/drive/M
y Drive/', 'sentiment_dataset_test_400.tar.gz',
                                       mode = 'test', path_to
_saved_embeddings = '/content/drive/My Drive/word2vec/')
```

```
# Create custom training/validation dataloader
train_data_loader = torch.utils.data.DataLoader(train_dataset
_400, batch_size=1, shuffle=True, num_workers=1)
test_data_loader = torch.utils.data.DataLoader(test_dataset_4
00, batch_size=1, shuffle=True, num_workers=1)

# Initialize GRU with Embeddings
device = torch.device('cuda') if torch.cuda.is_available() el
se torch.device('cpu')
print(f"Device: {device}")

model = GRUnetWithEmbeddings(input_size=300, hidden_size=100,
output_size=2, num_layers=2)


epochs = 2
display_interval = 500

# Count the number of learnable parameters
number_of_learnable_params = sum(p.numel() for p in model.par
ameters() if p.requires_grad)
print("\nThe number of learnable parameters in the model:", n
umber_of_learnable_params)

# Count the number of layers
num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model:", num_layers)

# training for sentiment_dataset_train_400
net1_losses = run_code_for_training_for_text_classification_w
ith_GRU(device, model, dataloader=train_data_loader,
                                        model_name='HW9_extra
_400', epochs=epochs, display_interval=display_interval)
```

```
# testing for sentiment_dataset_train_400
save_path = '/content/drive/My Drive/saved_models/HW9_extra_4
00.pt'
run_code_for_testing_text_classification_with_GRU(device, mod
el, dataloader=test_data_loader,
                                display_interval=display_interva
l, model_path=save_path, model_name='HW9_extra_400.pt')

# Initialize BiGRU with Embeddings
device = torch.device('cuda') if torch.cuda.is_available() el
se torch.device('cpu')
print(f"Device: {device}")

model = BiGRUnetWithEmbeddings(input_size=300, hidden_size=10
0, output_size=2, num_layers=2)



epochs = 2
display_interval = 500

# Count the number of learnable parameters
number_of_learnable_params = sum(p.numel() for p in model.par
ameters() if p.requires_grad)
print("\nThe number of learnable parameters in the model:", n
umber_of_learnable_params)

# Count the number of layers
num_layers = len(list(model.parameters()))
print("\nThe number of layers in the model:", num_layers)

# training for sentiment_dataset_train_400
net1_losses = run_code_for_training_for_text_classification_w
ith_GRU(device, model, dataloader=train_data_loader,
                                          model_name='HW9_extra
_bi400', epochs=epochs, display_interval=display_interval)
```

```
# training for sentiment_dataset_train_400
save_path = '/content/drive/My Drive/saved_models/HW9_extra_b
i400.pt'
run_code_for_testing_text_classification_with_GRU(device, mod
el, dataloader=test_data_loader,
                                display_interval=display_interva
l, model_path=save_path, model_name='HW9_extra_bi400.pt')
```