RNN Implementation for Sentiment Analysis and Classification

1. Introduction to Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to recognize patterns in sequences of data, such as text, stock marked prices, audio etc. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs, which makes them ideal for such tasks.

However, RNNs have a significant issue known as the vanishing gradient problem. This issue arises during the training process, where the gradients of the loss function tend to get smaller and smaller as they are propagated back in time. This results in earlier layers learning very slowly, or not at all, making it difficult for the network to learn long-range dependencies in the data.

To overcome the vanishing gradient problem, gating techniques such as Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRUs) are used. These techniques introduce gates that control the flow of information in and out of the memory component of the recurrent units. This allows the network to effectively learn to keep or discard information over long sequences, mitigating the impact of the vanishing gradient problem and enabling the network to learn long-range dependencies.

2. Creating word embedding for Sentiment Analysis tasks:

Word embeddings play a pivotal role in natural language processing (NLP), particularly in sentiment analysis tasks. Sentiment analysis aims to determine the sentiment expressed in a piece of text, whether it's positive, negative, or neutral. Word embeddings, which represent words as dense vectors in a continuous vector space, capture semantic meanings and contextual relationships between words. Leveraging word embeddings in sentiment analysis allows models to understand the nuanced meaning of words and phrases, leading to more accurate sentiment classification.

Our approach to implementing sentiment analysis with word embeddings is by utilizing pre-trained transformer-based models like DistilBERT. DistilBERT, a smaller and faster variant of BERT (Bidirectional Encoder Representations from Transformers), provides contextualized word embeddings by considering the entire input text. The DistilBertTokenizer is used to preprocess text inputs, breaking them into tokens and converting them into input IDs compatible with the DistilBERT model. DistilBERTModel then generates contextualized word embeddings for each tokenized input, and provides us with torch tensor representations of those words.

For the task at hand we were provided with a data file, with sentences and corresponding sentiments(Positive, Negative and Neutral). For the sentence data, for preprocessing we passed it through two tokenizers. One, a custom tokenizer based on full words and another the DistilBertTokenizer, based on subwords, then passing the tokenized inputs through DistilBERTModel to obtain embeddings, and then feeding these embeddings into the GRU based neural network for sentiment classification. The dataset class create below outputs embedding based on the complete word as well as the sub word, but we have used the sub word based embedding as the input to the model.

The code to implement the same using the Pytorch dataset and dataloader classes can be found below:

```
class SentencesDataset(Dataset):
    def __init__(self, file_path):
      self.df = pd.read_csv(file_path)

      # Tokenize and create word embedding for sentences
      sentences = [i for i in self.df['Sentence']]
      word_tokenized_sentences = [sentence.split() for sentence in sentences]
      max_len = max([len(sentence) for sentence in word_tokenized_sentences])
```

```
      padded_sentences = [sentence + ['[PAD]'] * (max_len - len(sentence)) for sentence i
n word_tokenized_sentences]
      self.tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')

      model_ckpt = "distilbert-base-uncased"
      distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
      bert_tokenized_sentences_ids = [ distilbert_tokenizer.encode(sentence , padding ='m
ax_length',truncation =True ,max_length = max_len )for sentence in sentences ]

            #Vocabulary created, by appding the sentence to normalize the size of the inp
ut
      vocab = {}
      vocab ['[PAD]'] = 0

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

      padded_sentences_ids = [[vocab[token] for token in sentence] for sentence in padded
_sentences]

      distilbert_model = DistilBertModel.from_pretrained(model_ckpt)
            # Word Embedding creation based on the tokizer implemented
      word_embeddings = []
      count_1 = 0
      for tokens in padded_sentences_ids :
        input_ids = torch.tensor(tokens).unsqueeze(0)
        with torch . no_grad ():
          outputs = distilbert_model(input_ids)
          count_1 += 1
          print(count_1)
        word_embeddings.append(outputs.last_hidden_state)

      subword_embeddings = []
      count_2 = 0
      for tokens in bert_tokenized_sentences_ids :
        input_ids = torch.tensor(tokens).unsqueeze(0)
        with torch . no_grad ():
          outputs = distilbert_model(input_ids)
          count_2 += 1
          print(count_2)
        subword_embeddings.append(outputs.last_hidden_state)

      self.embedding1 = word_embeddings
      self.embedding2 = subword_embeddings

      # Map sentiment labels to one-hot vectors
      self.sentiment_map = {'positive': [1, 0, 0],
                            'neutral': [0, 1, 0],
                            'negative': [0, 0, 1]}
```

```
    def __len__(self):
      return len(self.df)

    def __getitem__(self, idx):
      word_embedding = self.embedding1[idx]
      subword_embedding = self.embedding2[idx]
      sentiment = self.df.iloc[idx]['Sentiment']

      # Convert sentiment label to one-hot vector
      sentiment_label = self.sentiment_map[sentiment]
      sentiment_tensor = torch.tensor(sentiment_label)

      return word_embedding, subword_embedding, sentiment_tensor
```

```
# Creating and saving the dataset, and creating corresponding train and test dataloaders
sentences_dataset = SentencesDataset('/content/drive/MyDrive/HW9/data.csv')
torch.save(sentences_dataset, '/content/drive/MyDrive/HW9/SentencesDataset.pt')
dataset = torch.load('/content/drive/MyDrive/HW9/SentencesDataset.pt')
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [0.8, 0.2])
train_dataloader = DataLoader(train_dataset, batch_size=1, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=1, shuffle=True)
```

Distribution of the classes for provided dataset:

Sentiment	Count of Sentiment
neutral	3130
positive	1852
negative	860
Grand Total	**5842**

Table 1

2.1 GRU:

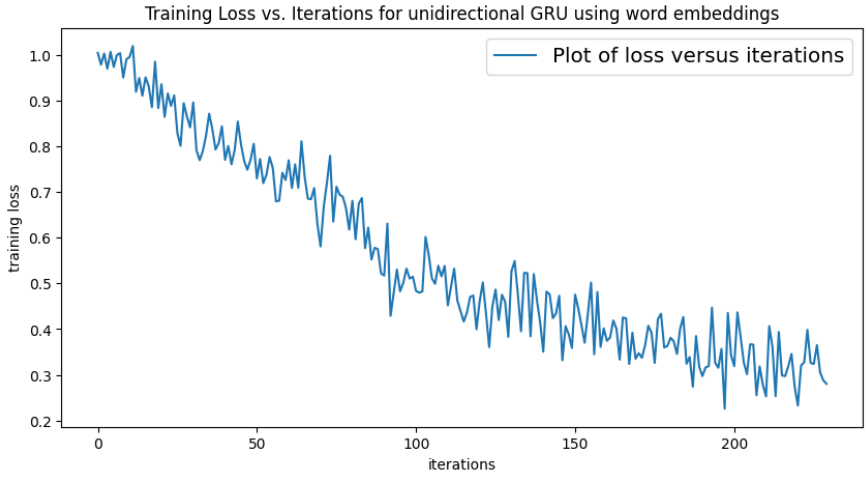
Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem often encountered in traditional RNNs. GRU models are characterized by their gating mechanisms, which regulate the flow of information within the network. Unlike traditional RNNs, which have separate memory and output units, GRUs merge these functionalities into a single unit. The key components of a GRU unit include the reset gate and update gate. The reset gate determines how much past information to forget, while the update gate controls how much new information to incorporate into the current state. These gates enable GRUs to selectively update their hidden states, allowing them to capture long-range dependencies in sequential data while mitigating the vanishing gradient problem.

Model implementation code for the GRU can be found below:

```
# Has been implemented as per the DLStudio Library as provided by Prof. Kak
class GRUnet(nn.Module):
```

```
            def __init__(self, input_size, hidden_size, output_size, num_layers, drop_pro
b=0.2):
                super(GRUnet, self).__init__()
                self.hidden_size = hidden_size
                self.num_layers = num_layers
                self.gru = nn.GRU(input_size, hidden_size, num_layers)
                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
```

Training Results:

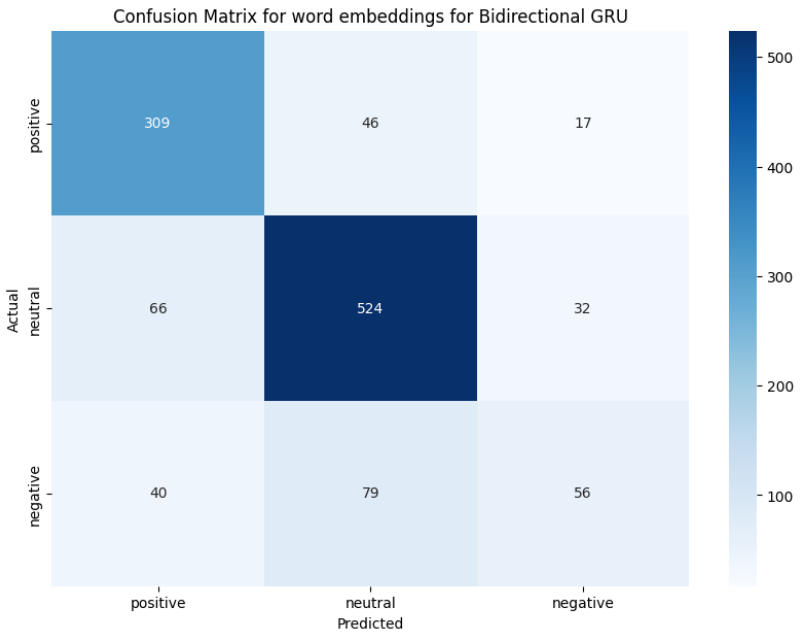


Training Result 1

Testing Result:

```
Overall classification accuracy: 76.05%

Number of negative reviews tested: 175

Number of neutral reviews tested: 622

Number of positive reviews tested: 372
```

Accuracy 1



Confusion Matrix 1

2.2 Bi-GRU:

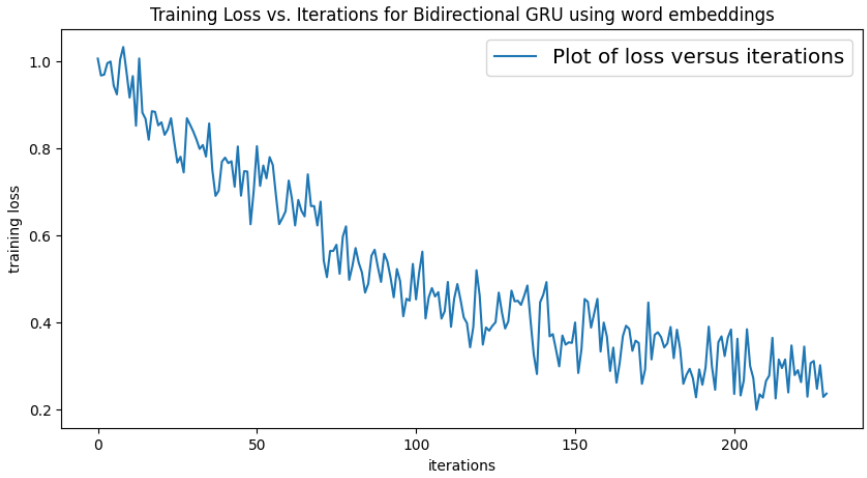
Bidirectional Gated Recurrent Unit (GRU) models are an extension of the standard GRU architecture that enables the network to process input sequences in both forward and backward directions. Introduced as an enhancement to traditional unidirectional RNNs, bidirectional GRUs leverage information from past and future time steps simultaneously, allowing the model to capture dependencies from both directions.In bidirectional GRU models, the input sequence is fed into two separate GRU layers: one processing the sequence in the forward direction and the other in the backward direction. Each layer computes hidden states independently, incorporating information from preceding and succeeding time steps, respectively. The outputs of the two layers are typically concatenated or combined in some way to produce the final output sequence.

Model Implementation code for bidirectional GRU can be found below:

```
# Implemented with few changes to Prof. Kak's DL Studio implmentation of GRU model.
class BiGRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, drop_prob=0.2):
```

```
        super(BiGRUnet, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = True  # Set bidirectional to True for bidirectional GRU
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True, bidirect
ional=True)
        self.fc = nn.Linear(hidden_size * 2, output_size)  # Multiply by 2 for bidirectio
nal GRU
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        # Concatenate the hidden states from both directions
        out = torch.cat((out[:, -1, :self.hidden_size], out[:, 0, self.hidden_size:]), di
m=1)
        out = self.fc(self.relu(out))
        out = self.logsoftmax(out)
        return out, h

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

Training Result:

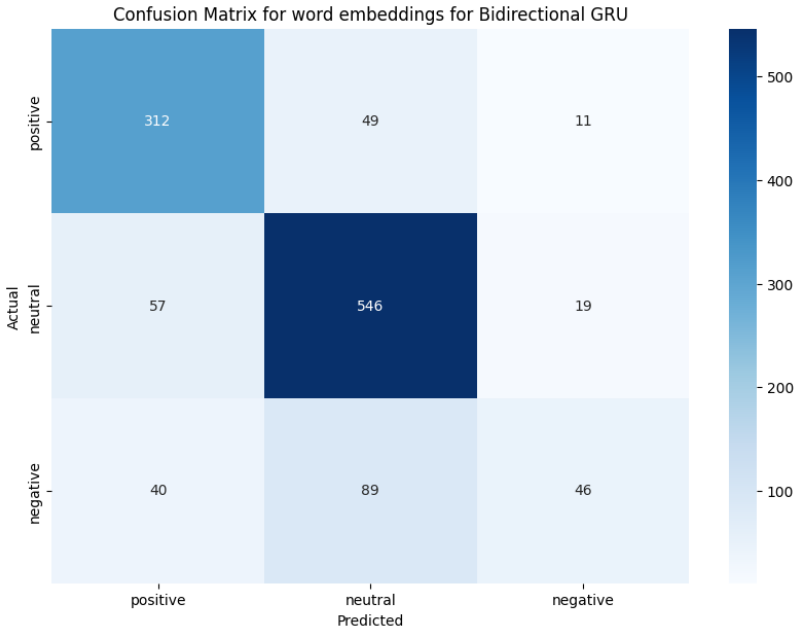


Training Result 2

Testing Result:

```
Overall classification accuracy: 77.33%

Number of negative reviews tested: 175

Number of neutral reviews tested: 622

Number of positive reviews tested: 372
```

Accuracy 2



Confusion Matrix 2

The test result for Bi Directional GRU is marginally better than the performance of normal GRU as can be seen from the classification accuracy mentioned in the images Accuracy 1 and Accuracy 2, that seems to be due to the fact that that the Bi directional GRU is capturing the context a little bit better than the normal GRU.

2.3 Comparison between the performances of GRU and BiGRU Models:

- Bidirectional GRU models have an advantage over normal GRU models in capturing long-range dependencies in sequential data. By processing input sequences in both forward and backward directions, bidirectional GRUs can capture context from both past and future time steps, enabling them to better understand the overall sequence structure as well as the capture the contextual information from the training better than a normal GRU.
- Hence, in tasks that require sequential understanding of the information, bi-directional GRUs tend to perform better than normal GRUs.
- For the case implemented above, we can see that performance(Images: Training Result, Accuracy and Confusion Matrix 1 and 2) of both normal GRU and Bidirectional GRU is quite mediocre(~76% for GRU and ~77% for BiGru) even after running the code for 10 epochs. This seems to primarily be due to class imbalance in the data, where

the neutral reviews are much higher in number than the positive and negative classes(Table 1). But the performance of bidirectional GRU is marginally better than that of GRU, due to the reasons mentioned in the previous points.

- Even for the extra credit case, we can see that the performance is good(Images: EC Training Result, EC Accuracy and EC Confusion Matrix both 1 and 2 and their sub parts, in the Extra Credit section below) but still seem to be somewhat affected by the issue of class imbalance. But we can see that the overall performance of the BiGRU model is marginally better than the normal GRU in this case too.
- Also affecting the performance seems to be the size of the training data, for the 400 Dataset, we can see that the performance for both GRU and BiGRU is better than the 200 dataset. Hence, we can see that the performance of the model improved when the size of the training dataset is larger.
- From the two implementations in this assignment, what I can understand, the issues in performance stem from the class imbalance in the datasets, as well as the size of the datasets. To improve the same, we can try increasing the size of the dataset and try to resolve the issue of class imbalance using a weighted loss function, oversampling the minority classes or under sampling the majority classes.

3. Implementation on Movie Review Dataset:

FurtherI have implemented the above models on the extracted data sets from the link provided for the movie sentiment dataset.

For extracting the data and creating the corresponding dataset and dataloader, I have used a class similar to the one mentioned in the DL Studio module using the word2vec embeddings:

```
# implemented in a similar manner as in the DL Studio module
class SentimentAnalysisDataset(torch.utils.data.Dataset):
            def __init__(self, train_or_test, dataset_file, path_to_saved_embeddings=Non
e):
                super(SentimentAnalysisDataset, self).__init__()
                import gensim.downloader as gen_api
#                self.word_vectors = gen_api.load("word2vec-google-news-300")
                self.path_to_saved_embeddings = path_to_saved_embeddings
                self.train_or_test = train_or_test
                root_dir = '/teamspace/studios/this_studio/Extra_Credit/data/'
                f = gzip.open(root_dir + dataset_file, 'rb')
                dataset = f.read()
                if path_to_saved_embeddings is not None:
                    import gensim.downloader as genapi
                    from gensim.models import KeyedVectors
                    if os.path.exists(path_to_saved_embeddings + 'vectors.kv'):
                        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings +
'vectors.kv')
                    else:
                        print("""\n\nSince this is your first time to install the word2ve
c embeddings, it may take"""
                              """\na couple of minutes. The embeddings occupy around 3.6G
B of your disk space.\n\n""")
                        self.word_vectors = genapi.load("word2vec-google-news-300")
                        ##  'kv' stands for  "KeyedVectors", a special datatype used by g
ensim because it
```

```
                        ##  has a smaller footprint than dict
                        self.word_vectors.save(path_to_saved_embeddings + 'vectors.kv')
                if train_or_test == 'train':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_train, self.negative_reviews_train, self.vo
cab = pickle.loads(dataset, encoding='latin1')
                    else:
                        self.positive_reviews_train, self.negative_reviews_train, self.vo
cab = pickle.loads(dataset)
                    self.categories = sorted(list(self.positive_reviews_train.keys()))
                    self.category_sizes_train_pos = {category : len(self.positive_reviews
_train[category]) for category in self.categories}
                    self.category_sizes_train_neg = {category : len(self.negative_reviews
_train[category]) for category in self.categories}
                    self.indexed_dataset_train = []
                    for category in self.positive_reviews_train:
                        for review in self.positive_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 1])
                    for category in self.negative_reviews_train:
                        for review in self.negative_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 0])
                    random.shuffle(self.indexed_dataset_train)
                elif train_or_test == 'test':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_test, self.negative_reviews_test, self.voca
b = pickle.loads(dataset, encoding='latin1')
                    else:
                        self.positive_reviews_test, self.negative_reviews_test, self.voca
b = pickle.loads(dataset)
                    self.vocab = sorted(self.vocab)
                    self.categories = sorted(list(self.positive_reviews_test.keys()))
                    self.category_sizes_test_pos = {category : len(self.positive_reviews_
test[category]) for category in self.categories}
                    self.category_sizes_test_neg = {category : len(self.negative_reviews_
test[category]) for category in self.categories}
                    self.indexed_dataset_test = []
                    for category in self.positive_reviews_test:
                        for review in self.positive_reviews_test[category]:
                            self.indexed_dataset_test.append([review, category, 1])
                    for category in self.negative_reviews_test:
                        for review in self.negative_reviews_test[category]:
                            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(embedding))
                    else:
```

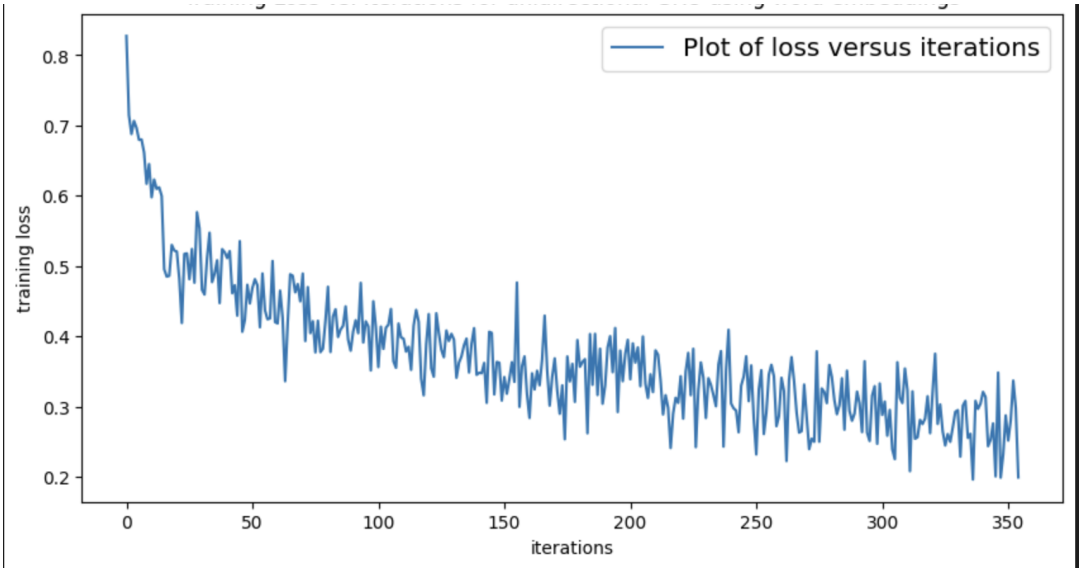
```
                        next
#                review_tensor = torch.FloatTensor( list_of_embeddings )
                review_tensor = torch.FloatTensor( np.array(list_of_embeddings) )
                return review_tensor

            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.train_or_test == 'train':
                    return len(self.indexed_dataset_train)
                elif self.train_or_test == 'test':
                    return len(self.indexed_dataset_test)

            def __getitem__(self, idx):
                sample = self.indexed_dataset_train[idx] if self.train_or_test == '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_sentiment)
                review_tensor = self.review_to_tensor(review)
                category_index = self.categories.index(review_category)
                sample = {'review'       : review_tensor,
                          'category'     : category_index, # should be converted to tenso
r, but not yet used
                          'sentiment'    : review_sentiment }
                return sample
```

3.1 GRU:

The network used here is the same as the one mentioned in the first GRU section.

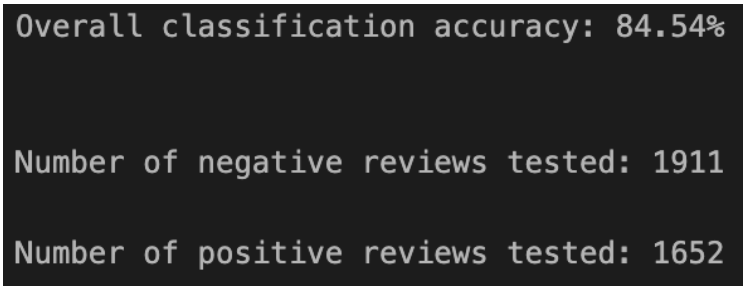
3.1.1. Sentiment Dataset 400

Training Result:

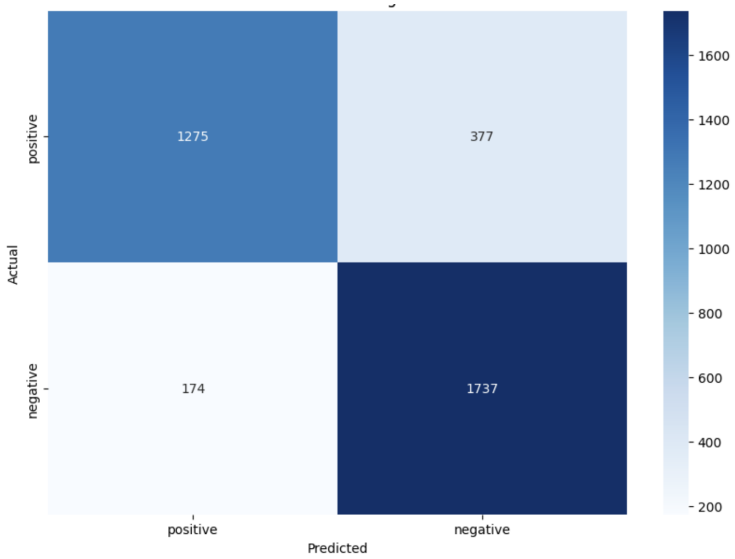


EC Training Result 1.1

Testing Result:



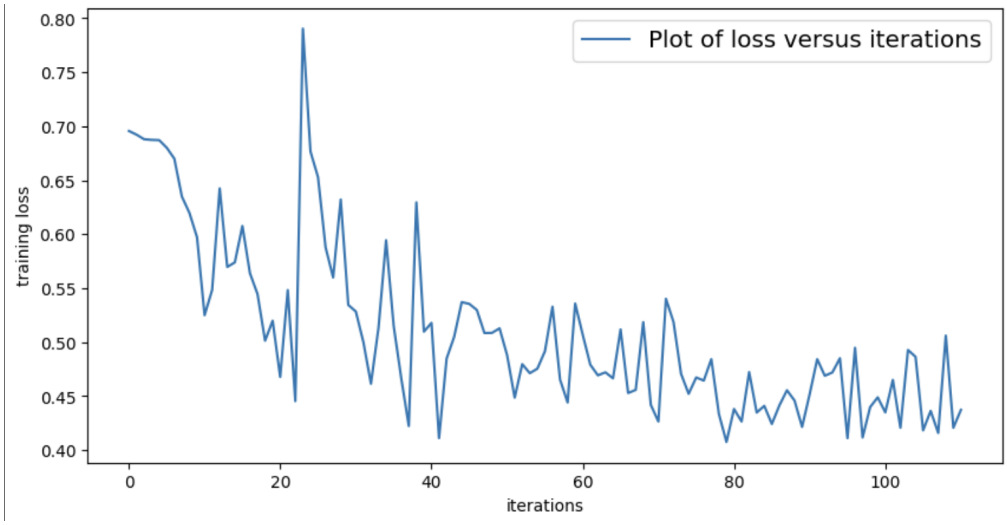
EC Accuracy 1.1



EC Confusion Matrix 1.1

3.1.2. Sentiment Dataset 200

Training Result:

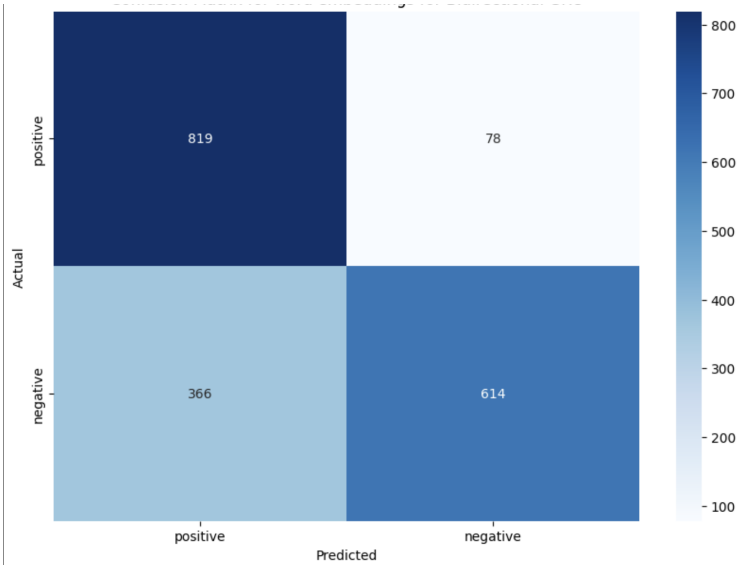


EC Training result 1.2

Testing Result:

```
Overall classification accuracy: 76.35%

Number of negative reviews tested: 980

Number of positive reviews tested: 897
```

EC Accuracy 1.2



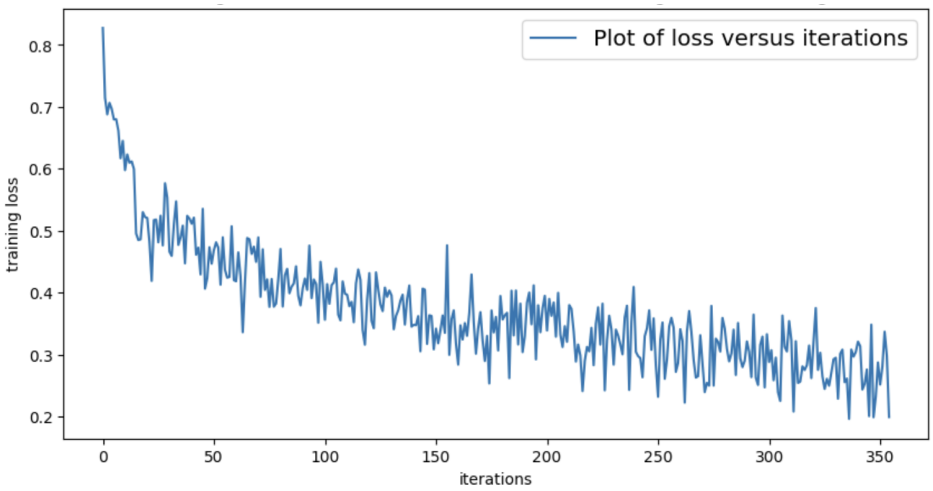
EC Confusion Matrix 1.2

3.2 Bi-GRU

The network used here is the same as the one mentioned in the first Bi-directional GRU section.

3.2.1 Sentiment Dataset 400

Training Result:

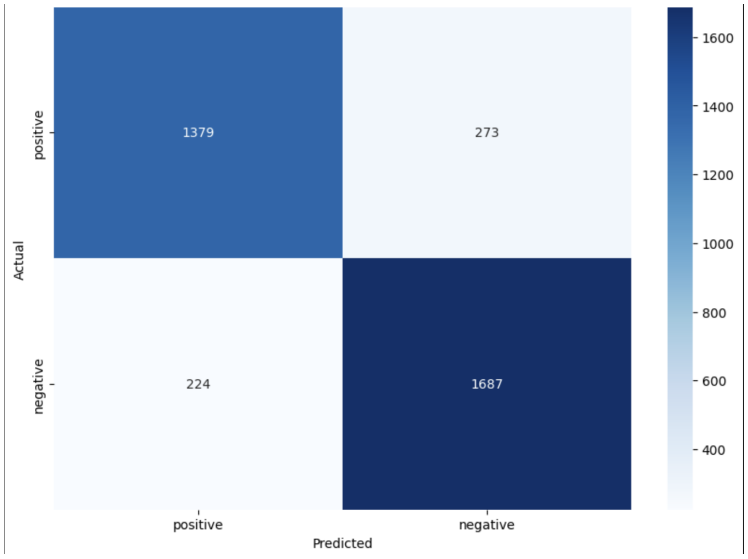


EC Training Result 2.1

Testing Result:

```
Overall classification accuracy: 86.05%

Number of negative reviews tested: 1911

Number of positive reviews tested: 1652
```

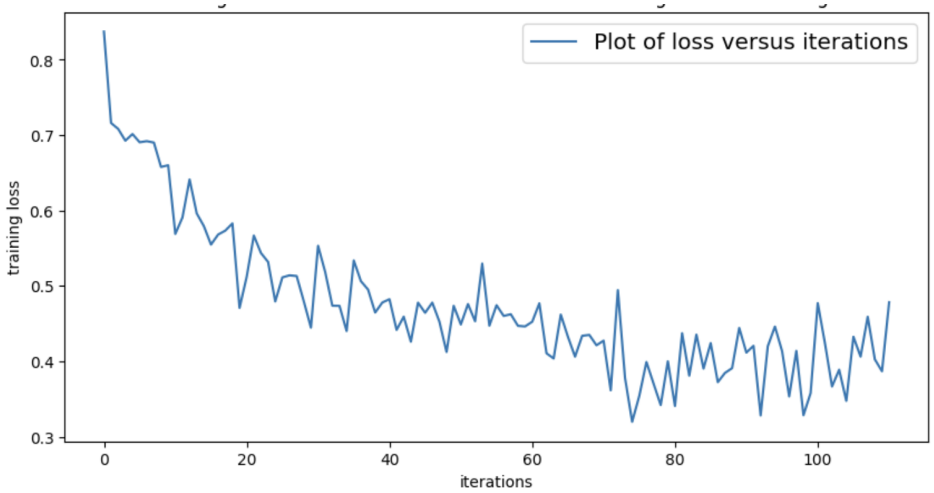
EC Accuracy 2.1



EC Confusion Matrix 2.1

3.2.2 Sentiment Dataset 200

Training Result:

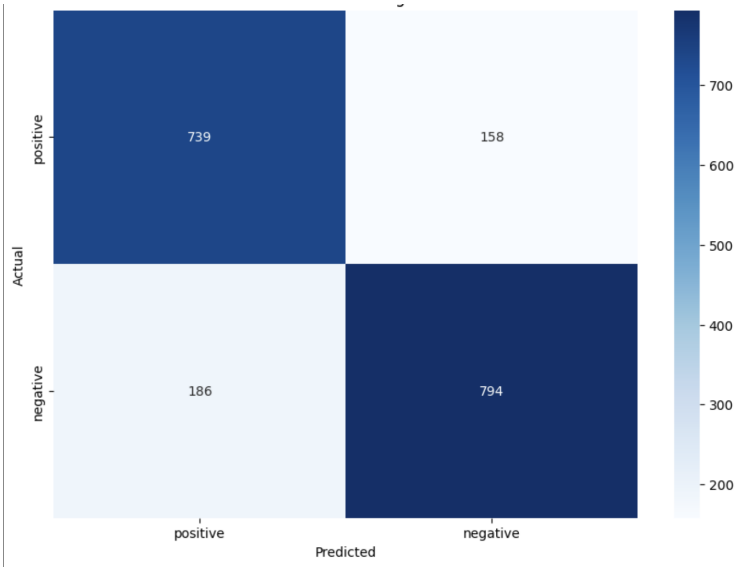


EC Training Result 2.2

Testing Result:

```
Overall classification accuracy: 81.67%

Number of negative reviews tested: 980

Number of positive reviews tested: 897
```

Ec Accuracy 2.2



EC Confusion Matrix 2.2

Source Code:

```
import torch
from torch.utils.data import Dataset, DataLoader, RandomSampler
from transformers import DistilBertTokenizer, DistilBertModel
import pandas as pd
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torch.optim as optim
import numpy as np
import numbers
import re
import math
import random
import copy
import matplotlib.pyplot as plt
import gzip
import pickle
import seaborn as sns
# import pymsgbox
import time
import logging

class SentencesDataset(Dataset):
    def __init__(self, file_path, test_train):
      self.df = pd.read_csv(file_path)
```

```
      # Tokenize and create word embedding for sentences
      sentences = [i for i in self.df['Sentence']]
      if test_train == 'train':
        sentences = sentences[:(4675)]
      elif test_train == 'test':
        sentences = sentences[4676:5843]
      else:
        raise ValueError("Invalid test_train value. Must be 'train' or 'test'.")
      word_tokenized_sentences = [sentence.split() for sentence in sentences]
      max_len = max([len(sentence) for sentence in word_tokenized_sentences])
      padded_sentences = [sentence + ['[PAD]'] * (max_len - len(sentence)) for sentence i
n word_tokenized_sentences]
      self.tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')

      model_ckpt = "distilbert-base-uncased"
      distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
      bert_tokenized_sentences_ids = [ distilbert_tokenizer.encode(sentence , padding ='m
ax_length',truncation =True ,max_length = max_len )for sentence in sentences ]

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

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

      padded_sentences_ids = [[vocab[token] for token in sentence] for sentence in padded
_sentences]

      distilbert_model = DistilBertModel.from_pretrained(model_ckpt)

      word_embeddings = []
      count_1 = 0
      for tokens in padded_sentences_ids :
        input_ids = torch.tensor(tokens).unsqueeze(0)
        with torch . no_grad ():
          outputs = distilbert_model(input_ids)
          count_1 += 1
          print(count_1)
        word_embeddings.append(outputs.last_hidden_state)

      subword_embeddings = []
      count_2 = 0
      for tokens in bert_tokenized_sentences_ids :
        input_ids = torch.tensor(tokens).unsqueeze(0)
        with torch . no_grad ():
          outputs = distilbert_model(input_ids)
          count_2 += 1
          print(count_2)
```

```
        subword_embeddings.append(outputs.last_hidden_state)

      self.embedding1 = word_embeddings
      self.embedding2 = subword_embeddings

      # Map sentiment labels to one-hot vectors
      self.sentiment_map = {'positive': [1, 0, 0],
                            'negative': [0, 1, 0],
                            'neutral': [0, 0, 1]}

    def __len__(self):
      return len(self.df)

    def __getitem__(self, idx):
      word_embedding = self.embedding1[idx]
      subword_embedding = self.embedding2[idx]
      sentiment = self.df.iloc[idx]['Sentiment']

      # Convert sentiment label to one-hot vector
      sentiment_label = self.sentiment_map[sentiment]
      sentiment_tensor = torch.tensor(sentiment_label)

      return word_embedding, subword_embedding, sentiment_tensor

train_dataset = SentencesDataset('/content/drive/MyDrive/HW9/data.csv', 'train')
torch.save(train_dataset, '/content/drive/MyDrive/HW9/dataset/train_SentencesDataset.pt')

test_dataset = SentencesDataset('/content/drive/MyDrive/HW9/data.csv', 'test')
torch.save(test_dataset, '/content/drive/MyDrive/HW9/dataset/testSentencesDataset.pt')

train_dataset = torch.load('/content/drive/MyDrive/HW9/dataset/train_SentencesDataset.p
t')
test_dataset = torch.load('/content/drive/MyDrive/HW9/dataset/testSentencesDataset.pt')

len(train_dataset)

len(test_dataset)

# train_random_sampler = RandomSampler(train_dataset, num_samples=int(0.8 * len(train_dat
aset) ))
# test_random_sampler = RandomSampler(test_dataset, num_samples=int(0.2 * len(test_datase
t) ))
# train_dataloader = DataLoader(train_dataset, batch_size=1, sampler=train_random_sample
r)
# test_dataloader = DataLoader(test_dataset, batch_size=1, sampler=test_random_sampler)

x, y, z = next(iter(train_dataloader))

x.shape

dataset = torch.load('/content/drive/MyDrive/HW9/SentencesDataset.pt')
```

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [0.8, 0.2])

train_dataloader = DataLoader(train_dataset, batch_size=1, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=1, shuffle=True)

class GRUnet(nn.Module):
            def __init__(self, input_size, hidden_size, output_size, num_layers, drop_pro
b=0.2):
                super(GRUnet, self).__init__()
                self.hidden_size = hidden_size
                self.num_layers = num_layers
                self.gru = nn.GRU(input_size, hidden_size, num_layers)
                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

class BiGRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, drop_prob=0.2):
        super(BiGRUnet, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = True  # Set bidirectional to True for bidirectional GRU
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True, bidirect
ional=True)
        self.fc = nn.Linear(hidden_size * 2, output_size)  # Multiply by 2 for bidirectio
nal GRU
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        # Concatenate the hidden states from both directions
        out = torch.cat((out[:, -1, :self.hidden_size], out[:, 0, self.hidden_size:]), di
m=1)
        out = self.fc(self.relu(out))
        out = self.logsoftmax(out)
        return out, h
```

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

def run_code_for_training_for_text_classification_with_GRU(net, display_train_loss=Fals
e):
            filename_for_out = "performance_numbers_" + str(1) + ".txt"
            FILE = open('/content/drive/MyDrive/HW9/saved_model'+filename_for_out, 'w')
            net.to(torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'))
            ##  Note that the GRUnet now produces the LogSoftmax output:
            criterion = nn.NLLLoss()
            accum_times = []
            # optimizer = optim.SGD(net.parameters(),
            #             lr=1e-3, momentum=0.9)
            optimizer = optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.999))
            start_time = time.perf_counter()
            training_loss_tally = []
            for epoch in range(1):
                print("")
                running_loss = 0.0
                for i, data in enumerate(train_dataloader):
                    review_tensor, bemb, sentiment = data
                    review_tensor = review_tensor[0]
                    sentiment = sentiment[0]
                    review_tensor = review_tensor.to(torch.device('cuda:0' if torch.cuda.
is_available() else 'cpu'))
                    sentiment = sentiment.to(torch.device('cuda:0' if torch.cuda.is_avail
able() else 'cpu'))
                    ## The following type conversion needed for MSELoss:
                    ##sentiment = sentiment.float()
                    optimizer.zero_grad()
                    hidden = net.init_hidden().to(torch.device('cuda:0' if torch.cuda.is_
available() else 'cpu'))
                    for k in range(review_tensor.shape[1]):
                        output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tenso
r[0,k],0),0), hidden)
                    # output, hidden = net(review_tensor, hidden)
                    ## If using NLLLoss, CrossEntropyLoss
                    # print(f'Output:{output}')
                    # print(f'Sentiment Argmax:{torch.argmax(sentiment.unsqueeze(0),1)}')
                    loss = criterion(output, torch.argmax(sentiment.unsqueeze(0),1))
                    ## If using MSELoss:
                    ## loss = criterion(output, sentiment)
                    running_loss += loss.item()
                    loss.backward()
                    optimizer.step()
```

```
                    if i % 10 == 9:
                        avg_loss1 = running_loss / float(10)
                    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: %.5
f" % (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_times))))
            print("\nFinished Training\n")
            return net.state_dict(), training_loss_tally, running_loss
            # torch.save(net.state_dict(), '/content/drive/MyDrive/HW9/saved_model/saved_
model_gru')
            # if display_train_loss:
            plt.figure(figsize=(10,5))
            plt.title("Training Loss vs. Iterations")
            plt.plot(training_loss_tally)
            plt.xlabel("iterations")
            plt.ylabel("training loss")
#           plt.legend()
            plt.legend(["Plot of loss versus iterations"], fontsize="x-large")
            plt.savefig("training_loss.png")
            plt.show()

model_gru = GRUnet(768, hidden_size=512, output_size=3, num_layers=2)
trained_net_gru, avg_loss_gru, running_loss_gru = run_code_for_training_for_text_classifi
cation_with_GRU(model_gru, display_train_loss=True)

plt.figure(figsize=(10,5))
plt.title("Training Loss vs. Iterations")
plt.plot(avg_loss_gru)
plt.xlabel("iterations")
plt.ylabel("training loss")
#plt.legend()
plt.legend(["Plot of loss versus iterations"], fontsize="x-large")
# plt.savefig("training_loss2.png")
plt.show()

torch.save(trained_net_gru, '/content/drive/MyDrive/HW9/dataset/saved_model_gru')

model_bigru = BiGRUnet(768, hidden_size=512, output_size=3, num_layers=2)
trained_net_bigru, avg_loss_bigru ,running_loss_bigru = run_code_for_training_for_text_cl
assification_with_GRU(model_bigru, display_train_loss=True) ### Issue is with the value o
f sentiment check out in the morning

torch.save(trained_net_bigru, '/content/drive/MyDrive/HW9/dataset/saved_model_bigru')
```

```
plt.figure(figsize=(10,5))
plt.title("Training Loss vs. Iterations")
plt.plot(avg_loss_bigru)
plt.xlabel("iterations")
plt.ylabel("training loss")
#plt.legend()
plt.legend(["Plot of loss versus iterations"], fontsize="x-large")
plt.savefig("training_loss.png")
plt.show()

def run_code_for_testing_text_classification_with_GRU(net, path):
            net.load_state_dict(torch.load(path))
            net.to(torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'))
            classification_accuracy = 0.0
            negative_total = 0
            positive_total = 0
            confusion_matrix = torch.zeros(3,3)
            with torch.no_grad():
                for i, data in enumerate(test_dataloader):
                    wemb,review_tensor,sentiment = data
                    review_tensor = review_tensor[0]
                    sentiment = sentiment[0]
                    review_tensor = review_tensor.to(torch.device('cuda:0' if torch.cuda.
is_available() else 'cpu'))
                    sentiment = sentiment.to(torch.device('cuda:0' if torch.cuda.is_avail
able() else 'cpu'))
                    hidden = net.init_hidden().to(torch.device('cuda:0' if torch.cuda.is_
available() else 'cpu'))
                    for k in range(review_tensor.shape[1]):
                        output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tenso
r[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  predicted     gt_label=%d
\n\n" % (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%%" %  (float(classification_a
ccuracy) * 100 /float(i)))
            out_percent = np.zeros((3,3), dtype='float')
            out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0] / float(negative_tot
al))
            out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1] / float(negative_tot
```

```
al))
            out_percent[0,2] = "%.3f" % (100 * confusion_matrix[0,2] / float(negative_tot
al))
            out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0] / float(positive_tot
al))
            out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1] / float(positive_tot
al))
            out_percent[1,2] = "%.3f" % (100 * confusion_matrix[1,2] / float(positive_tot
al))
            out_percent[2,0] = "%.3f" % (100 * confusion_matrix[2,0] / float(negative_tot
al))
            out_percent[2,1] = "%.3f" % (100 * confusion_matrix[2,1] / float(negative_tot
al))
            out_percent[2,2] = "%.3f" % (100 * confusion_matrix[2,2] / float(negative_tot
al))
            print("\n\nNumber of positive reviews tested: %d" % positive_total)
            print("\n\nNumber of negative reviews tested: %d" % negative_total)
            print("\n\nDisplaying the confusion matrix:\n")
            out_str = "                      "
            out_str +=  "%18s    %18s" % ('predicted negative', 'predicted positive')
            print(out_str + "\n")
            for i,label in enumerate(['true negative', 'true positive']):
                out_str = "%12s:  " % label
                for j in range(2):
                    out_str +=  "%18s" % out_percent[i,j]
                print(out_str)
            return confusion_matrix

output_gru = run_code_for_testing_text_classification_with_GRU(model_gru, '/content/driv
e/MyDrive/HW9/dataset/saved_model_gru')

sns.heatmap(output_gru, annot=True, cmap='Greens', fmt='g', xticklabels=['positive','nega
tive', 'neutral'], yticklabels=['positive','negative', 'neutral'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Heatmap')
plt.show()

output_bigru = run_code_for_testing_text_classification_with_GRU(model_bigru, '/content/d
rive/MyDrive/HW9/dataset/saved_model_bigru')

sns.heatmap(output_bigru, annot=True, cmap='Greens', fmt='g', xticklabels=['positive','ne
gative', 'neutral'], yticklabels=['positive','negative', 'neutral'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Heatmap')
plt.show()
```

Source code: Extra Credit

```
# %%
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

import seaborn as sns
import pandas as pd
import gzip
import pickle

# %%
device=torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")

# %%
class SentimentAnalysisDataset(torch.utils.data.Dataset):
            def __init__(self, train_or_test, dataset_file, path_to_saved_embeddings=Non
e):
                super(SentimentAnalysisDataset, self).__init__()
                import gensim.downloader as gen_api
#                self.word_vectors = gen_api.load("word2vec-google-news-300")
                self.path_to_saved_embeddings = path_to_saved_embeddings
                self.train_or_test = train_or_test
                root_dir = '/teamspace/studios/this_studio/Extra_Credit/data/'
                f = gzip.open(root_dir + dataset_file, 'rb')
                dataset = f.read()
                if path_to_saved_embeddings is not None:
                    import gensim.downloader as genapi
                    from gensim.models import KeyedVectors
                    if os.path.exists(path_to_saved_embeddings + 'vectors.kv'):
                        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings +
'vectors.kv')
                    else:
                        print("""\n\nSince this is your first time to install the word2ve
c embeddings, it may take"""
```

```
                                """\na couple of minutes. The embeddings occupy around 3.6G
B of your disk space.\n\n""")
                        self.word_vectors = genapi.load("word2vec-google-news-300")
                        ##  'kv' stands for  "KeyedVectors", a special datatype used by g
ensim because it
                        ##  has a smaller footprint than dict
                        self.word_vectors.save(path_to_saved_embeddings + 'vectors.kv')
                if train_or_test == 'train':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_train, self.negative_reviews_train, self.vo
cab = pickle.loads(dataset, encoding='latin1')
                    else:
                        self.positive_reviews_train, self.negative_reviews_train, self.vo
cab = pickle.loads(dataset)
                    self.categories = sorted(list(self.positive_reviews_train.keys()))
                    self.category_sizes_train_pos = {category : len(self.positive_reviews
_train[category]) for category in self.categories}
                    self.category_sizes_train_neg = {category : len(self.negative_reviews
_train[category]) for category in self.categories}
                    self.indexed_dataset_train = []
                    for category in self.positive_reviews_train:
                        for review in self.positive_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 1])
                    for category in self.negative_reviews_train:
                        for review in self.negative_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 0])
                    random.shuffle(self.indexed_dataset_train)
                elif train_or_test == 'test':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_test, self.negative_reviews_test, self.voca
b = pickle.loads(dataset, encoding='latin1')
                    else:
                        self.positive_reviews_test, self.negative_reviews_test, self.voca
b = pickle.loads(dataset)
                    self.vocab = sorted(self.vocab)
                    self.categories = sorted(list(self.positive_reviews_test.keys()))
                    self.category_sizes_test_pos = {category : len(self.positive_reviews_
test[category]) for category in self.categories}
                    self.category_sizes_test_neg = {category : len(self.negative_reviews_
test[category]) for category in self.categories}
                    self.indexed_dataset_test = []
                    for category in self.positive_reviews_test:
                        for review in self.positive_reviews_test[category]:
                            self.indexed_dataset_test.append([review, category, 1])
                    for category in self.negative_reviews_test:
                        for review in self.negative_reviews_test[category]:
                            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(embedding))
                    else:
                        next
#                review_tensor = torch.FloatTensor( list_of_embeddings )
                review_tensor = torch.FloatTensor( np.array(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.train_or_test == 'train':
                    return len(self.indexed_dataset_train)
                elif self.train_or_test == 'test':
                    return len(self.indexed_dataset_test)

            def __getitem__(self, idx):
                sample = self.indexed_dataset_train[idx] if self.train_or_test == '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_sentiment)
                review_tensor = self.review_to_tensor(review)
                category_index = self.categories.index(review_category)
                sample = {'review'       : review_tensor,
                          'category'     : category_index, # should be converted to tenso
r, but not yet used
                          'sentiment'    : review_sentiment }
                return sample

# %%
dataroot = "/teamspace/studios/this_studio/Extra_Credit/data/"


dataset_archive_train = "sentiment_dataset_train_400.tar.gz"
#dataset_archive_train = "sentiment_dataset_train_200.tar.gz"

dataset_archive_test =  "sentiment_dataset_test_400.tar.gz"
#dataset_archive_test = "sentiment_dataset_test_200.tar.gz"

path_to_saved_embeddings = "/teamspace/studios/this_studio/Extra_Credit/data/word2vec/"
```

```
#path_to_saved_embeddings = "./data/TextDatasets/word2vec/"

dataserver_train = SentimentAnalysisDataset(
                                train_or_test = 'train',
                                dataset_file = dataset_archive_train,
                                path_to_saved_embeddings = path_to_saved_embeddings,
                    )
dataserver_test = SentimentAnalysisDataset(
                                train_or_test = 'test',
                                dataset_file = dataset_archive_test,
                                path_to_saved_embeddings = path_to_saved_embeddings,
                   )


# %%
y = dataserver_train[10]

# %%
y['review'][0].shape

# %%
train_dataloader = torch.utils.data.DataLoader(dataserver_train,batch_size=1,shuffle=Tru
e)
test_dataloader = torch.utils.data.DataLoader(dataserver_test,batch_size=1,shuffle=False)

# %%
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_layers)
        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

# %%
```

```
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_layers, dropout=drop_prob, bidirec
tional=True)
        self.fc = nn.Linear(hidden_size * 2, output_size)  # Multiply by 2 for bidirectio
nal
        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:]), di
m=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

# %%
def run_code_for_training_with_GRU( net,train_dataloader,device, display_train_loss=Tru
e):
    filename_for_out = "performance_numbers_" + str(1) + ".txt"
    FILE = open(filename_for_out, 'w')
    net.to(device)
    ## Note that the TEXTnet and TEXTnetOrder2 both produce LogSoftmax output:
    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(3):
        print("")
        running_loss = 0.0
        for i, data in enumerate(train_dataloader):
            hidden = net.init_hidden().to(device)

            review_tensor,category,sentiment = data['review'], data['category'], data['se
ntiment']
            review_tensor = review_tensor.to(device)
```

```
            sentiment = sentiment.to(device)

            optimizer.zero_grad()
            hidden = net.init_hidden().to(device)
            for k in range(review_tensor.shape[1]):
               # input[0,:] = review_tensor[0,k]
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],
0),0), hidden)
            loss = criterion(output, torch.argmax(sentiment,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" % (ep
och+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_times))))
    print("\nFinished Training\n")
    torch.save(net.state_dict(), "/teamspace/studios/this_studio/Extra_Credit/data/unigru
_EC_200.pt")
    if display_train_loss:
        plt.figure(figsize=(10,5))
        plt.title("Training Loss vs. Iterations for unidirectional GRU using word embeddi
ngs")
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")
#                plt.legend()
        plt.legend(["Plot of loss versus iterations"], fontsize="x-large")
        plt.savefig("training_loss.png")
        plt.show()
    return training_loss_tally

# %%
def run_code_for_training_with_BiGRU( net,train_dataloader,device, display_train_loss=Tru
e):
    filename_for_out = "performance_numbers_" + str(1) + ".txt"
    FILE = open(filename_for_out, 'w')
    net.to(device)
    ## Note that the TEXTnet and TEXTnetOrder2 both produce LogSoftmax output:
    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(3):
        print("")
        running_loss = 0.0
        for i, data in enumerate(train_dataloader):


            review_tensor,category,sentiment = data['review'], data['category'], data['se
ntiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            optimizer.zero_grad()
            hidden = net.init_hidden().to(device)
            for k in range(review_tensor.shape[1]):
               # input[0,:] = review_tensor[0,k]
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],
0),0), hidden)
            loss = criterion(output, torch.argmax(sentiment,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" % (ep
och+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_times))))
    print("\nFinished Training\n")
    torch.save(net.state_dict(), "/teamspace/studios/this_studio/Extra_Credit/data/bigru_
EC_200.pt")
    if display_train_loss:
        plt.figure(figsize=(10,5))
        plt.title("Training Loss vs. Iterations for unidirectional GRU using word embeddi
ngs")
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")
#                plt.legend()
        plt.legend(["Plot of loss versus iterations"], fontsize="x-large")
        plt.savefig("training_loss.png")
        plt.show()
    return training_loss_tally
```

```
# %%
def run_code_for_testing_text_classification_with_GRU(net, test_dataloader, device, pat
h):
    net.load_state_dict(torch.load(path))
    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(test_dataloader):
            review_tensor,category,sentiment = data['review'], data['category'], data['se
ntiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            hidden = net.init_hidden().to(device)

            for k in range(review_tensor.shape[1]):
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(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, pre
dicted_idx, gt_idx))

            if predicted_idx == gt_idx:
                classification_accuracy += 1

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


            confusion_matrix[gt_idx, predicted_idx] += 1

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

    out_percent = np.zeros((2,2), dtype='float')
    out_percent[0,:] = 100 * confusion_matrix[0,:] / negative_total
    out_percent[1,:] = 100 * confusion_matrix[1,:] / positive_total

    print("\n\nNumber of negative reviews tested: %d" % negative_total)
    print("\nNumber of positive reviews tested: %d" % positive_total)

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

```
    plt.figure(figsize=(10, 7))
    sns.heatmap(confusion_matrix, annot=True, fmt='g', cmap='Blues', xticklabels=['positi
ve', 'negative',],
                yticklabels=['positive', 'negative'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix for word embeddings for Bidirectional GRU')
    plt.show()

# %%
model_gru = GRUnet(300, hidden_size=100, output_size=2, num_layers=2)
device=torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
number_of_learnable_params = sum(p.numel() for p in model_gru.parameters() if p.requires_
grad)

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

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

# %%
avg_loss_uni1_EC= run_code_for_training_with_GRU(net=model_gru,train_dataloader=train_dat
aloader, device=device, display_train_loss=True)

# %%
model_bigru = BiGRUnet(300, hidden_size=100, output_size=3, num_layers=3)
device=torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
number_of_learnable_params = sum(p.numel() for p in model_bigru.parameters() if p.require
s_grad)

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

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

# %%
avg_loss_biuni1_EC= run_code_for_training_with_BiGRU(net=model_bigru,train_dataloader=tra
in_dataloader, device=device, display_train_loss=True)

# %%
run_code_for_testing_text_classification_with_GRU(net=model_gru,test_dataloader=test_data
loader, device=device, path='/teamspace/studios/this_studio/Extra_Credit/data/unigru_EC.p
t')

# %%
run_code_for_testing_text_classification_with_GRU(net=model_bigru,test_dataloader=test_da
taloader, device=device, path='/teamspace/studios/this_studio/Extra_Credit/data/bigru_EC.
```

```
pt')

# %%
dataroot = "/teamspace/studios/this_studio/Extra_Credit/data/"


dataset_archive_train_200 = "sentiment_dataset_train_200.tar.gz"
#dataset_archive_train = "sentiment_dataset_train_200.tar.gz"

dataset_archive_test_200 =  "sentiment_dataset_test_200.tar.gz"
#dataset_archive_test = "sentiment_dataset_test_200.tar.gz"

path_to_saved_embeddings = "/teamspace/studios/this_studio/Extra_Credit/data/word2vec/"
#path_to_saved_embeddings = "./data/TextDatasets/word2vec/"

dataserver_train_200 = SentimentAnalysisDataset(
                                train_or_test = 'train',
                                dataset_file = dataset_archive_train_200,
                                path_to_saved_embeddings = path_to_saved_embeddings,
                   )
dataserver_test_200 = SentimentAnalysisDataset(
                                train_or_test = 'test',
                                dataset_file = dataset_archive_test_200,
                                path_to_saved_embeddings = path_to_saved_embeddings,
                  )

# %%
train_dataloader_200 = torch.utils.data.DataLoader(dataserver_train_200,batch_size=1,shuf
fle=True)
test_dataloader_200 = torch.utils.data.DataLoader(dataserver_test_200,batch_size=1,shuffl
e=False)

# %%
avg_loss_uni1_EC_200= run_code_for_training_with_GRU(net=model_gru,train_dataloader=train
_dataloader_200, device=device, display_train_loss=True)

# %%
avg_loss_biuni1_EC_200= run_code_for_training_with_BiGRU(net=model_bigru,train_dataloader
=train_dataloader_200, device=device, display_train_loss=True)

# %%
run_code_for_testing_text_classification_with_GRU(net=model_gru,test_dataloader=test_data
loader_200, device=device, path='/teamspace/studios/this_studio/Extra_Credit/data/unigru_
EC_200.pt')

# %%
run_code_for_testing_text_classification_with_GRU(net=model_bigru,test_dataloader=test_da
taloader_200, device=device, path='/teamspace/studios/this_studio/Extra_Credit/data/bigru
_EC_200.pt')

# %%
```