

HOMEWORK 9

Alexandre Olivé Pellicer

PREPARING THE DATA

In order to create the dataset we have used the code snippets provided in the instructions. We created a dictionary containing all the preprocessed data that we will use for training and testing everytime so that we just need to do the preprocessing once. We stored the dictionary in a pickle file. The dictionary is composed of 3 keys:

- Sentiments: The value is a subdictionary where the keys are integers and the values are one-hot vectors representing the 3 different sentiments
- Word embeddings: The value is a subdictionary where the keys are integers and the values are the word embeddings corresponding to the words of each sentence. The number of words in a sentence has been fixed to 81 so that some padding has been added. The embedding dimensions are 768.
- Subword embeddings: The value is a subdictionary where the keys are integers and the values are the subword embeddings corresponding to the subwords of each sentence. We have used the definition of subword specified in the instructions. The number of subwords in a sentence has been fixed to 81 so that some padding has been added. The embedding dimensions are 768.

The elements of the 3 subdictionaries are ordered so that the sentiment corresponding to key "1" is the one associated with the word embedding corresponding to key "1" and with the subword embedding corresponding to key "1".

Each of the 3 subdictionaries has a length of 5842 which is the total number of sentence-sentiment pairs contained in the "data.csv" file. We have created two new dictionaries stored in two different pickle files from the original dictionary described above. The first one contains 80% of the samples and it will be used for training. The second one contains the remaining 20% of the samples and it will be used for testing.

MODEL

In order to build the model and establish the training and evaluation logic to test the model we have been inspired by the implementation from DLStudio. This is a basic block diagram of our model which contains an instance of the nn.GRU() class as specified in the instructions:



Fig 1: Block diagram of the model used

For training the model we have used the Adam optimizer with a learning rate equal to 0.001. In order to compute the loss we have used the `nn.NLLLoss()` since our last layer of the model is a LogSoftmax Layer.

The input to our model is a tensor representing a sentence and a tensor to initialize the hidden states of the GRU module. The tensor representing a sentence is of shape `[81, 1, 768]` that indicates `[sequence_length, batch_size, input_size]`.

The objective of the model is to predict a one-hot vector of dimension 3 so that we can get which sentiment is associated with each sentence. There are 3 possible sentiments: “positive”, “neutral” and “negative”.

GRU Unidirectional

First of all, we implement a unidirectional GRU. That is that the output of the model will only depend on previous samples so that only the information a priori is used. For this reason, we initialize our model as:

```
model = SentimentGRU(input_size=768, hidden_size=128, output_size=3)
```

and the input tensor of shape `[sequence_length=81, batch_size=1, input_size=768]`. The initial hidden state tensor is set automatically so that is of size `[num_layers=3, batch_size=1, hidden_size=128]`

We define the GRU module inside our model as:

```
self.gru = nn.GRU(input_size, hidden_size, num_layers)
```

where we set `hidden_size = 128` and `num_layers = 3`. `Num_layers` equal to 3 means that in reality we don't have just 1 GRU cell but we have 3 concatenated GRU cells.

GRU Bidirectional

A possible approach to improve performance is to not only use the information a priori but also use the information a posteriori. In order to do this the size of the hidden states of the GRU module is doubled since we move from 1 state cell the receives the information from the previous state cell to two state cells one receiving the information from the previous state cell (as before) and the other receiving the information from the next state cell. Thus, the information a priori and a posteriori is used in order to make a prediction.

In terms of implementation this means that instead of initializing the hidden state tensor of shape `[num_layers=3, batch_size=1, hidden_size=128]` we do it with a tensor of shape `[num_layers=3*2, batch_size=1, hidden_size=128]`. Furthermore the output of the GRU module will be of doubled size. Therefore, only the first half of the output of the GRU module will be fed into the ReLU.

In order to indicate that we use the bidirectional implementation, we define the GRU module as:

```
self.gru = nn.GRU(input_size, hidden_size, num_layers,
bidirectional=True)
```

EXPERIMENTS

In this section we explain the experiments we have done using the data provided in the “data.csv” file. We have experimented with both types of tokenization presented in the instructions: word level and subword level. We have also experimented with both types of implementation of the GRU module: unidirectional GRU and bidirectional GRU. We analyze the performance of each implementation of the GRU module for each type of tokenization. The best performance achieved has an accuracy of **72.35%**. This accomplishes the requirement set by the TA on Piazza to have an accuracy of 70%.

Subword level tokenization

These are the training losses obtained when using the unidirectional and bidirectional implementations:

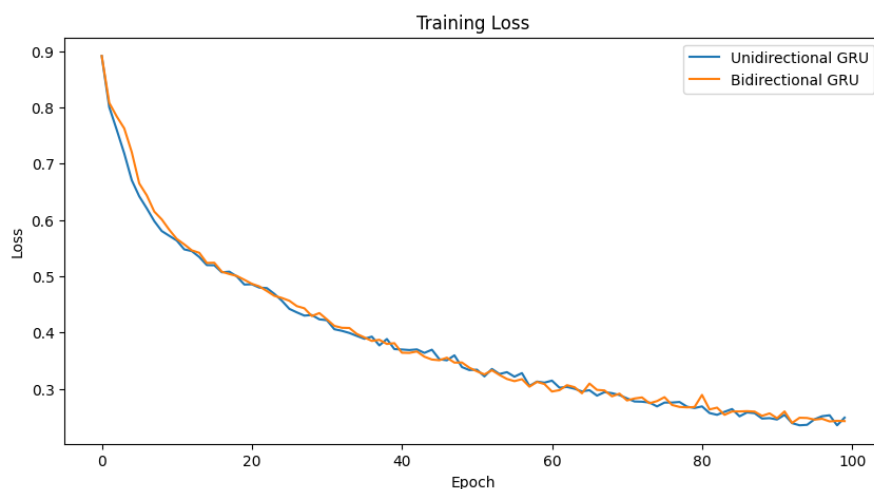


Fig 2: Plot containing the training loss of the unidirectional GRU and the bidirectional GRU

These are the accuracies obtained when testing the performance of the unidirectional and bidirectional implementations:

Accuracy of unidirectional GRU	Accuracy of bidirectional GRU
70.95%	72.35%

Fig 3: Table with accuracies

The best accuracy that we have been able to achieve after trying different configurations of the GRU with different optimizers and trying different word embeddings is 72.35% which is above the 70% stated as minimum.

These are the confusion matrixes obtained when testing the performance of the unidirectional and bidirectional implementations:

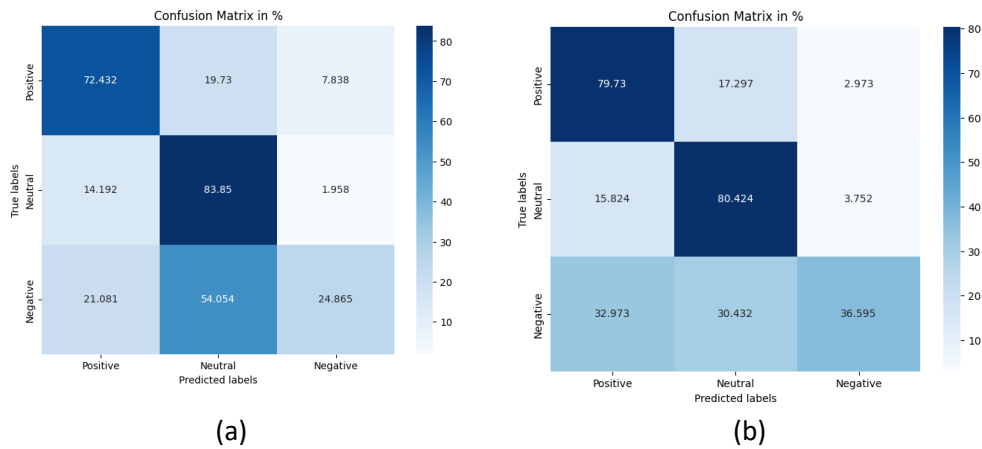


Fig 4: Confusion matrixes of (a) unidirectional GRU (b) bidirectional GRU

Commentaries (Does using a bidirectional scan make a difference in terms of test performance?):

Looking at the accuracy and confusion matrixes we can clearly state that the best performance is achieved when we use the bidirectional implementation. We can say this because the accuracy of the bidirectional implementation is higher and because the confusion matrix of the bidirectional implementation has higher values in the diagonal.

Going more into detail we see that both training losses converge towards a minimum and that when it comes to label sentences, the model struggles labeling sentences as negative.

Word level tokenization

These are the training losses obtained when using the unidirectional and bidirectional implementations:

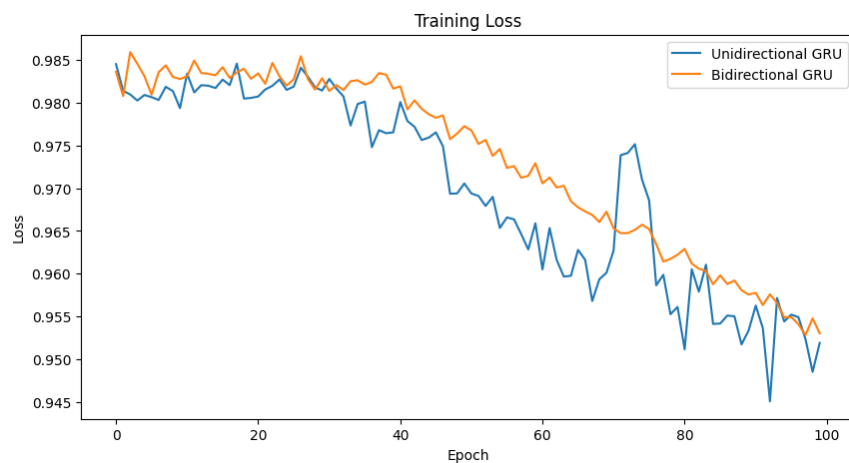


Fig 5: Plot containing the training loss of the unidirectional GRU and the bidirectional GRU

These are the accuracies obtained when testing the performance of the unidirectional and bidirectional implementations:

Accuracy of unidirectional GRU	Accuracy of bidirectional GRU
54.16%	55.61%

Fig 6: Table with accuracies

These are the confusion matrixes obtained when testing the performance of the unidirectional and bidirectional implementations:

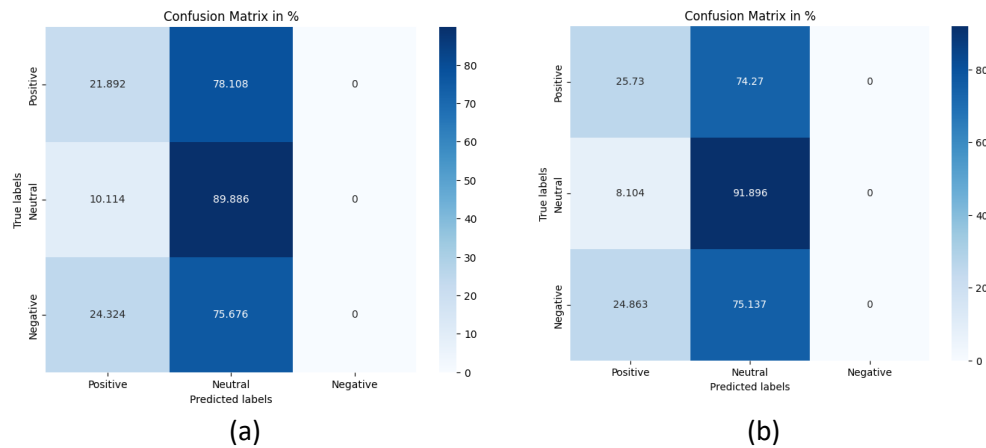


Fig 7: Confusion matrixes of (a) unidirectional GRU (b) bidirectional GRU

Commentaries (Does using a bidirectional scan make a difference in terms of test performance?):

In this case it is difficult to mention which implementation of the GRU works better because both of them work badly. The accuracies are low. If we had to choose an option, we would say that the bidirectional implementation works better because it has higher accuracy. This is again probably thanks to using information a priori and a posteriori in order to make the prediction.

From this experiment we can conclude that the word level tokenizer does not perform well. In the word level tokenizer, each word is treated as a token so that the list of tokens once tokenized all the dataset is short and during evaluation it will be hard for the model to deal with words it hasn't seen during training. In the subword level tokenizer, words are divided into fragments and each fragment is treated as a token. Thus, at the end there is more variety of tokens and maybe during testing the model may have to deal with sentences that contain words that it hasn't seen during training, but it would have seen fragments of these words. This is probably why the subword level tokenizer works better than the word level tokenizer.

Looking at the training losses it looks like they are trying to converge but that the amount of epochs that would be needed is very high (more than 100).

EXTRACREDIT

For the extracredit assignment we are asked to use the unidirectional and bidirectional implementation of the GRU presented before with two different datasets: sentiment_dataset_train_200 and sentiment_dataset_train_400. These datasets have been downloaded and we have used the SentimentAnalysisDataset class from DLStudio in order to manage the training and testing data efficiently during training and testing.

The training logic is the same as used in the previous section. Therefore, we have used the Adam optimizer with a learning rate equal to 0.001. In order to compute the loss we have used the nn.NLLLoss() since our last layer of the model is a LogSoftmax Layer.

In this case, since in these datasets the sentences are just labeled as “negative” or “positive” we have set the output of the networks to 2 since now we want to predict a one-hot vector of size 2. We also did some minor changes in the training logic in order to get adapted to this change.

sentiment dataset train 200

These are the training losses obtained when using the unidirectional and bidirectional implementations:

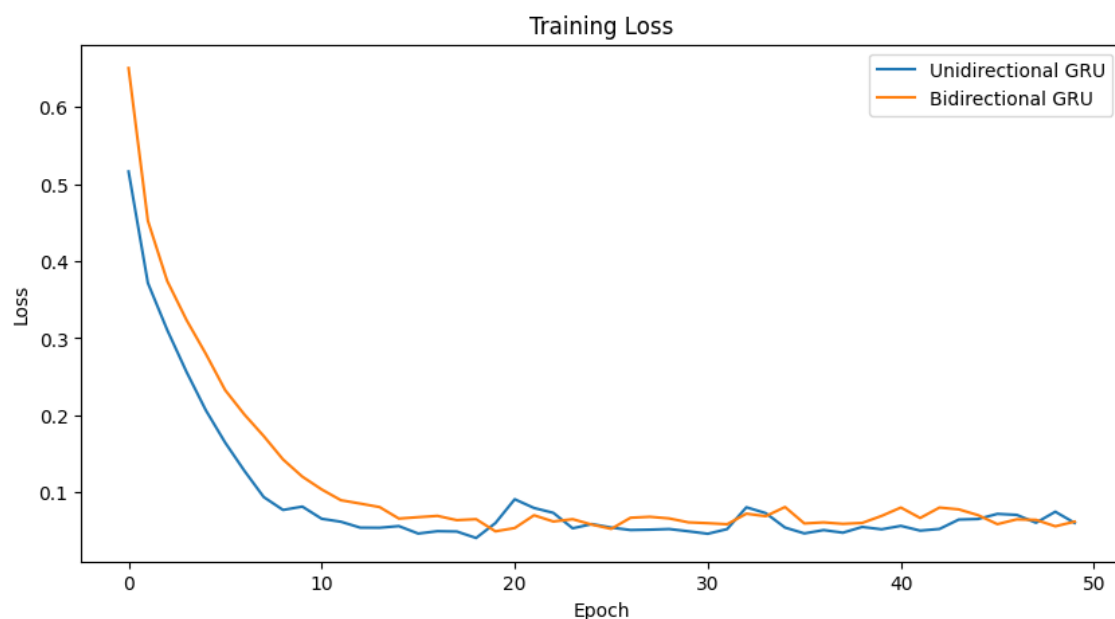


Fig 8: Plot containing the training loss of the unidirectional GRU and the bidirectional GRU

These are the accuracies obtained when testing the performance of the unidirectional and bidirectional implementations:

Accuracy of unidirectional GRU	Accuracy of bidirectional GRU
84.97%	88.38%

Fig 9: Table with accuracies

These are the confusion matrixes obtained when testing the performance of the unidirectional and bidirectional implementations:

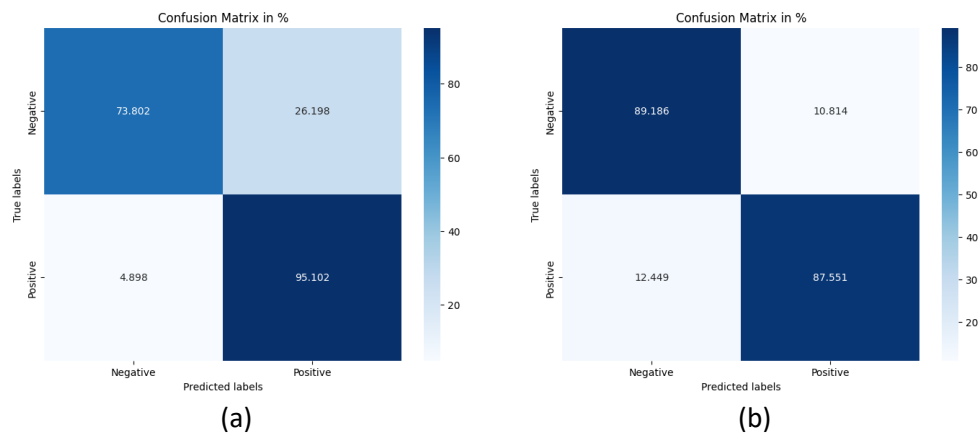


Fig 10: Confusion matrixes of (a) unidirectional GRU (b) bidirectional GRU

Commentaries (Does using a bidirectional scan make a difference in terms of test performance?):

Looking at the accuracy and confusion matrixes we can clearly state that the best performance is achieved when we use the bidirectional implementation. We can say this because the accuracy of the bidirectional implementation is higher and because the confusion matrix of the bidirectional implementation has higher values in the diagonal.

Going more into detail we see that both training losses converge towards a minimum and that when it comes to label sentences, the unidirectional model struggles labeling sentences as negative.

sentiment dataset train 400

These are the training losses obtained when using the unidirectional and bidirectional implementations:

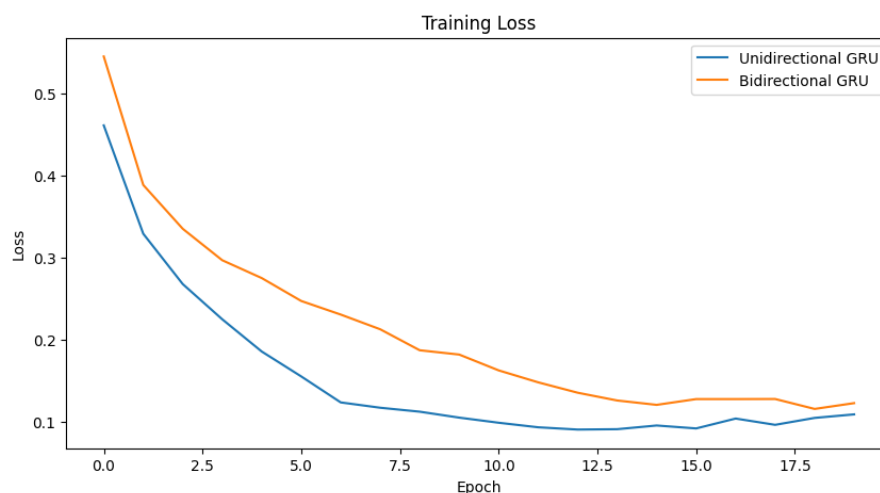


Fig 11: Plot containing the training loss of the unidirectional GRU and the bidirectional GRU

These are the accuracies obtained when testing the performance of the unidirectional and bidirectional implementations:

Accuracy of unidirectional GRU	Accuracy of bidirectional GRU
86.83%	88.01%

Fig 12: Table with accuracies

These are the confusion matrixes obtained when testing the performance of the unidirectional and bidirectional implementations:

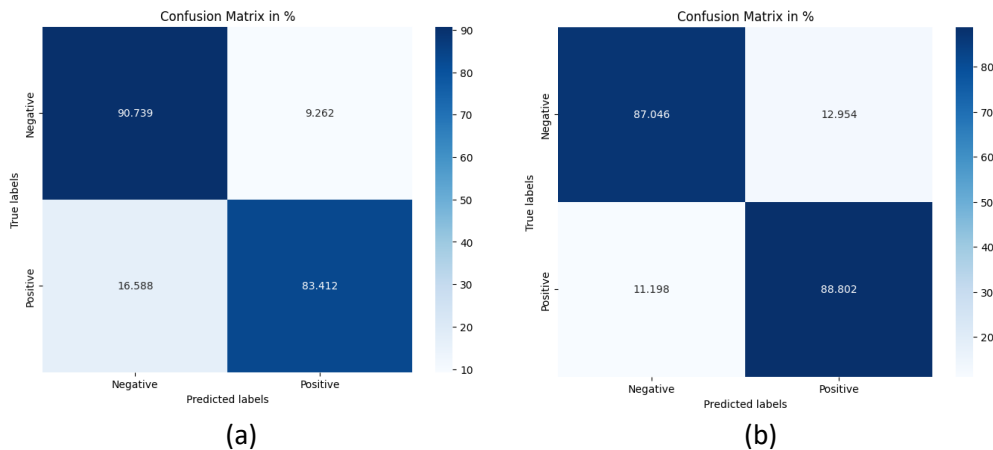


Fig 13: Confusion matrixes of (a) unidirectional GRU (b) bidirectional GRU

Commentaries (Does using a bidirectional scan make a difference in terms of test performance?):

Looking at the accuracy and confusion matrixes we can clearly state that the best performance is achieved when we use the bidirectional implementation. We can say this because the accuracy of the bidirectional implementation is higher and because the confusion matrix of the bidirectional implementation has higher values in the diagonal.

Going more into detail we see that both training losses converge towards a minimum and that when it comes to label sentences, the unidirectional GRU model struggles labeling sentences as positive.

Comparing performance on both datasets:

We see that in both cases the best performance is achieved when using the bidirectional GRU. The best performance is achieved when using the sentiment_dataset_train_200. It must be said that when using the sentiment_dataset_train_200 dataset we have trained the model for more epochs than when using the sentiment_dataset_train_400 dataset. This is because the sentiment_dataset_train_400 is heavier because the embeddings are of bigger dimension. Thus, training using this dataset also takes more time. Due to the limitations of the hardware available for us, we haven't been able to train for more epochs. Looking at the trend of the training loss when using the sentiment_dataset_train_400 dataset we can probably guess that it may be reduced a bit more so that the small gap between the best accuracy when using the sentiment_dataset_train_200 dataset and the sentiment_dataset_train_400 dataset may get closer. Anyway, 100% is not possible and in both cases the accuracy is very close to 100.

Comparing performance on the dataset data.csv:

We can see that when using the sentiment_dataset_train_200 and sentiment_dataset_train_400 datasets the performance is higher than when using the data.csv dataset. There may be 2 reasons to justify this behavior:

1. sentiment_dataset_train_200 and sentiment_dataset_train_400 contain more training samples. Thus, the model sees more variety of tokens during training so it may perform better during evaluation.
2. In sentiment_dataset_train_200 and sentiment_dataset_train_400 there are only 2 classes while in data.csv there are 3 classes. Thus, the chances of making a mistake in sentiment_dataset_train_200 and sentiment_dataset_train_400 is lower (50%) compared to in data.csv (66.66%)

CODE

Dataset.py

```
import torch
import sys
import random
import pickle
import csv
from transformers import DistilBertModel
from transformers import DistilBertTokenizer
import math

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

print(sentences)
print(sentiments)

## Get one-hot vectors for the sentiments -----
def sentiment_to_tensor(sentiment):
    sentiment_tensor = torch.zeros(3)
    if sentiment == "positive":
        sentiment_tensor[0] = 1
    elif sentiment == "neutral":
        sentiment_tensor[1] = 1
    elif sentiment == "negative":
        sentiment_tensor[2] = 1
    sentiment_tensor = sentiment_tensor.type(torch.long)
    return sentiment_tensor

encoded_sentiments = []
for sentiment in sentiments:
    encoded_sentiments.append(sentiment_to_tensor(sentiment))

## Word level tokenization -----
--
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 in word_tokenized_sentences]

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

## Subord level tokenization -----
----
model_ckpt = "distilbert-base-uncased"
distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)

bert_tokenized_sentences_ids = [distilbert_tokenizer.encode(sentence,
padding='max_length', truncation =True, max_length = max_len) for
sentence in sentences ]

bert_tokenized_sentences_tokens =
[distilbert_tokenizer.convert_ids_to_tokens(sentence) for sentence in
bert_tokenized_sentences_ids]

##Extracting embeddings -----
-----
model_name = 'distilbert/distilbert-base-uncased'
distilbert_model = DistilBertModel.from_pretrained(model_name)
# extract word embeddings
# we will use the last hidden state of the model
# you can use the other hidden states if you want
# the last hidden state is the output of the model
# after passing the input through the model

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)

# 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[1].shape)

## Saving embeddings and one-hot vectors in training and testing
dictionaries -----
dictionary = {}
for i in range(len(subword_embeddings)):
    dictionary[i] = {}
    dictionary[i]["sentiment"] = encoded_sentiments[i]
    dictionary[i]["word_embeddings"] = word_embeddings[i]
    dictionary[i]["subword_embeddings"] = subword_embeddings[i]

# Calculate the number of keys for each dictionary
total_keys = len(dictionary)
first_dict_keys = math.ceil(0.8 * total_keys) # 80% of total keys
second_dict_keys = total_keys - first_dict_keys # Remaining 20% of total
keys

# Create two new dictionaries
first_dictionary = {}
second_dictionary = {}

# Populate the first dictionary with 80% of the keys
for i in range(first_dict_keys):
    first_dictionary[i] = dictionary[i]

# Populate the second dictionary with the remaining 20% of the keys
for i in range(first_dict_keys, total_keys):
    second_dictionary[i - first_dict_keys] = dictionary[i]

with open('training_dictionary.pickle', 'wb') as f:
    pickle.dump(first_dictionary, f)

with open('testing_dictionary.pickle', 'wb') as f:
    pickle.dump(second_dictionary, f)

```

unidirectional.py

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import torch.optim as optim

```

```

import torch.nn as nn
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.nn.functional as F
import pickle

device = "cuda:2"

## Dataset class -----
class SentimentDataset(Dataset):
    def __init__(self, train_or_test, word_or_subword):
        self.train_or_test = train_or_test
        self.word_or_subword = word_or_subword

        with open('training_dictionary.pickle', 'rb') as f:
            self.training_dictionary = pickle.load(f)

        # Load the second dictionary from the pickle file
        with open('testing_dictionary.pickle', 'rb') as f:
            self.testing_dictionary = pickle.load(f)

    def __len__(self):
        length = 0
        if self.train_or_test == "train":
            length = len(self.training_dictionary)
        elif self.train_or_test == "test":
            length = len(self.testing_dictionary)
        return length

    def __getitem__(self, i):
        if self.train_or_test == "train":
            if self.word_or_subword == "word":
                sentence = self.training_dictionary[i]["word_embeddings"]
                sentence = sentence.permute(1, 0, 2)
            elif self.word_or_subword == "subword":
                sentence =
self.training_dictionary[i]["subword_embeddings"]
                sentence = sentence.permute(1, 0, 2)
            label = self.training_dictionary[i]["sentiment"]

        elif self.train_or_test == "test":
            if self.word_or_subword == "word":
                sentence = self.testing_dictionary[i]["word_embeddings"]
                sentence = sentence.permute(1, 0, 2)
            elif self.word_or_subword == "subword":
                sentence =
self.testing_dictionary[i]["subword_embeddings"]

```

```

        sentence = sentence.permute(1, 0, 2)
        label = self.testing_dictionary[i]["sentiment"]
        return sentence, label.type(torch.float)

# GRU model (based on DLStudio implementation)-----
-----
class SentimentGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
num_layers=3):
        super(SentimentGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers)
        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_size
        hidden =
weight.new( 3,          1,          self.hidden_size  ).zero_()
        return hidden

## Training loss
criterion = nn.NLLLoss()

## Optimizer
learning_rate = 0.001
model = SentimentGRU(input_size=768, hidden_size=128,
output_size=3).to(device) # 3 output classes
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

## Train model
def train_model(model, dataloader, criterion, optimizer, num_epochs):
    model.train()
    losses = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in dataloader:
            inputs = inputs.squeeze(0).to(device)
            labels = labels.to(device)

```

```

        optimizer.zero_grad()

        hidden = model.init_hidden().to(device)
        for k in range(inputs.shape[1]):
            output, hidden =
model(torch.unsqueeze(torch.unsqueeze(inputs[0,k],0),0), hidden)
            loss = criterion(output, torch.argmax(labels, 1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
        epoch_loss = running_loss / len(dataloader)
        losses.append(epoch_loss)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss}")

# Save generator model and loss plot
torch.save(model.state_dict(), './model_word_100_epochs.pth')

dictionary_losses = {}

nombre_imagen = 'yes'
dictionary_losses[nombre_imagen] = {
    'criterion1': losses
}

with open('/home/aolivepe/ECE60146/HW9/loss_word_100_epochs.pkl',
'wb') as archivo:
    pickle.dump(dictionary_losses, archivo)

# Plot training loss
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss over Epochs')
plt.savefig('training_loss_plot.png')
plt.show()

## Evaluate model
def evaluate_model(model, dataloader):
    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(dataloader):
            inputs, labels = data
            inputs = inputs.squeeze(0).to(device)
            labels = labels.to(device)

```

```

        hidden = model.init_hidden().to(device)
        for k in range(inputs.shape[1]):
            output, hidden =
model(torch.unsqueeze(torch.unsqueeze(inputs[0,k],0),0), hidden)
            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(labels).item()
            if i % 50 == 49:
                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:
                    positive_total += 1
                elif gt_idx == 1:
                    neutral_total += 1
                elif gt_idx == 2:
                    negative_total += 1
                confusion_matrix[gt_idx,predicted_idx] += 1
            print("\nOverall classification accuracy: %0.2f%"
% (float(classification_accuracy) * 100 /float(i)))
            out_percent = np.zeros((3,3), dtype='float')
            out_percent[0, 0] = "%.3f" % (100 * confusion_matrix[0, 0] /
float(positive_total))
            out_percent[0, 1] = "%.3f" % (100 * confusion_matrix[0, 1] /
float(positive_total))
            out_percent[0, 2] = "%.3f" % (100 * confusion_matrix[0, 2] /
float(positive_total))
            out_percent[1, 0] = "%.3f" % (100 * confusion_matrix[1, 0] /
float(neutral_total))
            out_percent[1, 1] = "%.3f" % (100 * confusion_matrix[1, 1] /
float(neutral_total))
            out_percent[1, 2] = "%.3f" % (100 * confusion_matrix[1, 2] /
float(neutral_total))
            out_percent[2, 0] = "%.3f" % (100 * confusion_matrix[2, 0] /
float(negative_total))
            out_percent[2, 1] = "%.3f" % (100 * confusion_matrix[2, 1] /
float(negative_total))
            out_percent[2, 2] = "%.3f" % (100 * confusion_matrix[2, 2] /
float(negative_total))
            print("\n\nNumber of positive reviews tested: %d" % positive_total)
            print("\n\nNumber of neutral reviews tested: %d" % neutral_total)
            print("\n\nNumber of negative reviews tested: %d" % negative_total)
            print("\n\nDisplaying the confusion matrix:\n")
            out_str = "
            out_str += "%18s %18s %18s" % ('predicted positive', 'predicted
neutral', 'predicted negative')
            print(out_str + "\n")

```

```

        for i,label in enumerate(['true positive', 'true neutral', 'true
negative']):
            out_str = "%12s: " % label
            for j in range(3):
                out_str += "%18s%" % out_percent[i,j]
            print(out_str)

## Define datasets and dataloaders
train_dataset =SentimentDataset("train", "word")
test_dataset =SentimentDataset("test", "word")

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

## Run training and evaluation
train_model(model, train_dataloader, criterion, optimizer,
num_epochs=100)
evaluate_model(model, test_dataloader)

```

unidirectional_extracredit.py

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import torch.optim as optim
import torch.nn as nn
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.nn.functional as F
import pickle
import gzip
import random
import sys
import os

device = "cuda:2"

## Dataset class (from DLStudio) -----
-----
class SentimentDataset(torch.utils.data.Dataset):
    def __init__(self, train_or_test, dataset_file,
path_to_saved_embeddings=None):
        super(SentimentDataset, self).__init__()

```

```

import gensim.downloader as gen_api
self.path_to_saved_embeddings = path_to_saved_embeddings
self.train_or_test = train_or_test
root_dir = "/"
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
word2vec embeddings, it may take""")
        """\na couple of minutes. The embeddings occupy
around 3.6GB of your disk space.\n\n""")
        self.word_vectors = genapi.load("word2vec-google-news-
300")
        ## 'kv' stands for "KeyedVectors", a special datatype
used by gensim 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.vocab = pickle.loads(dataset, encoding='latin1')
        else:
            self.positive_reviews_train, self.negative_reviews_train,
self.vocab = 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.vocab = pickle.loads(dataset, encoding='latin1')
            else:
                self.positive_reviews_test, self.negative_reviews_test,
self.vocab = 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 is ordinarily just a binary valued thing. It is 0 for
negative
        sentiment and 1 for positive sentiment. We need to pack this
value in a
        two-element tensor.
        """
        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 tensor, but not yet used
              'sentiment'  : review_sentiment }
    return sample

# GRU model (based on DLStudio implementation)-----
-----
class SentimentGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
num_layers=3):
        super(SentimentGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers)
        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_size
        hidden =
weight.new( 3,                1,                self.hidden_size        ).zero_()
        return hidden

## Training loss
criterion = nn.NLLLoss()

## Optimizer
learning_rate = 0.001
model = SentimentGRU(input_size=300, hidden_size=100,
output_size=2).to(device) # 3 output classes
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

## Train model
def train_model(model, dataloader, criterion, optimizer, num_epochs):
    model.train()
    losses = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(dataloader):
            review_tensor, category, sentiment = data['review'],
data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            optimizer.zero_grad()

            hidden = model.init_hidden().to(device)
            for k in range(review_tensor.shape[1]):
                output, hidden =
model(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0), hidden)
                loss = criterion(output, torch.argmax(sentiment, 1))
                running_loss += loss.item()
                loss.backward()
                optimizer.step()
            epoch_loss = running_loss / len(dataloader)
            losses.append(epoch_loss)
            print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss}")

    # Save generator model and loss plot
    torch.save(model.state_dict(),
'./model_100_epochs_extracredit_200.pth')

    dictionary_losses = {}

    nombre_imagen = 'yes'
    dictionary_losses[nombre_imagen] = {
        'criterion1': losses

```

```

    }

    with
open('/home/aolivepe/ECE60146/HW9/loss_100_epochs_extracredit_unidirectional_200.pkl', 'wb') as archivo:
    pickle.dump(dictionary_losses, archivo)

    # Plot training loss
    plt.plot(losses)
    plt.xlabel('Epoch')
    plt.ylabel('Training Loss')
    plt.title('Training Loss over Epochs')
    plt.savefig('training_loss_plot.png')
    plt.show()

## Evaluate model
def evaluate_model(model, dataloader):
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    neutral_total = 0
    confusion_matrix = torch.zeros(2,2)
    with torch.no_grad():
        for i, data in enumerate(dataloader):
            review_tensor, category, sentiment = data['review'],
            data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            hidden = model.init_hidden().to(device)
            for k in range(review_tensor.shape[1]):
                output, hidden =
model(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 % 50 == 49:
                    print("    [i=%4d]    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%"
% (float(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" % 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)

## Define datasets and dataloaders
dataset_archive_train =
"home/aolivepe/ECE60146/HW9/data/sentiment_dataset_train_200.tar.gz"

dataset_archive_test
= "home/aolivepe/ECE60146/HW9/data/sentiment_dataset_test_200.tar.gz"

path_to_saved_embeddings = "/home/aolivepe/ECE60146/HW9/"

train_dataset =SentimentDataset(
    train_or_test = 'train',
    dataset_file = dataset_archive_train,
    path_to_saved_embeddings =
path_to_saved_embeddings,
)
test_dataset =SentimentDataset(
    train_or_test = 'test',
    dataset_file = dataset_archive_test,
    path_to_saved_embeddings =
path_to_saved_embeddings,
)

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

```

```
## Run training and evaluation
train_model(model, train_dataloader, criterion, optimizer, num_epochs=20)
evaluate_model(model, test_dataloader)
```

bidirectional.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import torch.optim as optim
import torch.nn as nn
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.nn.functional as F
import pickle

device = "cuda:0"

## Dataset class -----
class SentimentDataset(Dataset):
    def __init__(self, train_or_test, word_or_subword):
        self.train_or_test = train_or_test
        self.word_or_subword = word_or_subword

        with open('training_dictionary.pickle', 'rb') as f:
            self.training_dictionary = pickle.load(f)

        # Load the second dictionary from the pickle file
        with open('testing_dictionary.pickle', 'rb') as f:
            self.testing_dictionary = pickle.load(f)

    def __len__(self):
        length = 0
        if self.train_or_test == "train":
            length = len(self.training_dictionary)
        elif self.train_or_test == "test":
            length = len(self.testing_dictionary)
        return length

    def __getitem__(self, i):
        if self.train_or_test == "train":
            if self.word_or_subword == "word":
                sentence = self.training_dictionary[i]["word_embeddings"]
                sentence = sentence.permute(1, 0, 2)
            elif self.word_or_subword == "subword":
```

```

        sentence =
self.training_dictionary[i]["subword_embeddings"]
        sentence = sentence.permute(1, 0, 2)
        label = self.training_dictionary[i]["sentiment"]

    elif self.train_or_test == "test":
        if self.word_or_subword == "word":
            sentence = self.testing_dictionary[i]["word_embeddings"]
            sentence = sentence.permute(1, 0, 2)
        elif self.word_or_subword == "subword":
            sentence =
self.testing_dictionary[i]["subword_embeddings"]
            sentence = sentence.permute(1, 0, 2)
            label = self.testing_dictionary[i]["sentiment"]
        return sentence, label.type(torch.float)

# GRU model (based on DLStudio implementation)-----
-----
class SentimentGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
num_layers=3):
        super(SentimentGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers,
bidirectional=True)
        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 = out[:, :, :128]
        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_size
        hidden =
weight.new( 6,          1,          self.hidden_size      ).zero_()
        return hidden

## Training loss
criterion = nn.NLLLoss()

## Optimizer

```

```

learning_rate = 0.001
model = SentimentGRU(input_size=768, hidden_size=128,
output_size=3).to(device) # 3 output classes
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train model
def train_model(model, dataloader, criterion, optimizer, num_epochs):
    model.train()
    losses = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in dataloader:
            inputs = inputs.squeeze(0).to(device)
            labels = labels.to(device)
            optimizer.zero_grad()

            hidden = model.init_hidden().to(device)
            for k in range(inputs.shape[1]):
                output, hidden =
model(torch.unsqueeze(torch.unsqueeze(inputs[0,k],0),0), hidden)
                loss = criterion(output, torch.argmax(labels, 1))
                running_loss += loss.item()
                loss.backward()
                optimizer.step()
            epoch_loss = running_loss / len(dataloader)
            losses.append(epoch_loss)
            print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss}")

    # Save generator model and loss plot
    torch.save(model.state_dict(),
'./model_bidirectional_word_100_epochs.pth')

    dictionary_losses = {}

    nombre_imagen = 'yes'
    dictionary_losses[nombre_imagen] = {
        'criterion1': losses
    }

    with
open('/home/aolivepe/ECE60146/HW9/loss_bidirectional_word_100_epochs.pkl'
, 'wb') as archivo:
        pickle.dump(dictionary_losses, archivo)

    # Plot training loss
    plt.plot(losses)
    plt.xlabel('Epoch')
    plt.ylabel('Training Loss')
    plt.title('Training Loss over Epochs')

```

```

plt.savefig('training_loss_plot.png')
plt.show()

## Evaluate your model
def evaluate_model(model, dataloader):
    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(dataloader):
            inputs, labels = data
            inputs = inputs.squeeze(0).to(device)
            labels = labels.to(device)

            hidden = model.init_hidden().to(device)
            for k in range(inputs.shape[1]):
                output, hidden =
model(torch.unsqueeze(torch.unsqueeze(inputs[0,k],0),0), hidden)
                predicted_idx = torch.argmax(output).item()
                gt_idx = torch.argmax(labels).item()
                if i % 50 == 49:
                    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:
                        positive_total += 1
                    elif gt_idx == 1:
                        neutral_total += 1
                    elif gt_idx == 2:
                        negative_total += 1
                    confusion_matrix[gt_idx, predicted_idx] += 1
                print("\nOverall classification accuracy: %0.2f%%"
% (float(classification_accuracy) * 100 / float(i)))
            out_percent = np.zeros((3,3), dtype='float')
            out_percent[0, 0] = "%.3f" % (100 * confusion_matrix[0, 0] /
float(positive_total))
            out_percent[0, 1] = "%.3f" % (100 * confusion_matrix[0, 1] /
float(positive_total))
            out_percent[0, 2] = "%.3f" % (100 * confusion_matrix[0, 2] /
float(positive_total))
            out_percent[1, 0] = "%.3f" % (100 * confusion_matrix[1, 0] /
float(neutral_total))
            out_percent[1, 1] = "%.3f" % (100 * confusion_matrix[1, 1] /
float(neutral_total))
            out_percent[1, 2] = "%.3f" % (100 * confusion_matrix[1, 2] /
float(neutral_total))

```

```

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

## Define datasets and dataloaders
train_dataset =SentimentDataset("train", "word")
test_dataset =SentimentDataset("test", "word")

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

## Run training and evaluation
train_model(model, train_dataloader, criterion, optimizer,
num_epochs=100)
evaluate_model(model, test_dataloader)

```

bidirectional_extracredit.py

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import torch.optim as optim
import torch.nn as nn
import torch
from torch.utils.data import Dataset

```

```

from torch.utils.data import DataLoader
import torch.nn.functional as F
import pickle
import gzip
import random
import sys
import os

device = "cuda:2"

## Dataset class (from DLStudio) -----
-----
class SentimentDataset(torch.utils.data.Dataset):
    def __init__(self, train_or_test, dataset_file,
path_to_saved_embeddings=None):
        super(SentimentDataset, self).__init__()
        import gensim.downloader as gen_api
        self.path_to_saved_embeddings = path_to_saved_embeddings
        self.train_or_test = train_or_test
        root_dir = "/"
        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
word2vec embeddings, it may take""")
                """\na couple of minutes. The embeddings occupy
around 3.6GB of your disk space.\n\n""")
                self.word_vectors = genapi.load("word2vec-google-news-
300")
                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.vocab = pickle.loads(dataset, encoding='latin1')
            else:
                self.positive_reviews_train, self.negative_reviews_train,
self.vocab = 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.vocab = pickle.loads(dataset, encoding='latin1')
        else:
            self.positive_reviews_test, self.negative_reviews_test,
self.vocab = 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 is ordinarily just a binary valued thing.  It is 0 for
negative
        sentiment and 1 for positive sentiment.  We need to pack this
value in a
        two-element tensor.
        """
        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 tensor, but not yet used
                  'sentiment'  : review_sentiment }
        return sample

# GRU model (based on DLStudio implementation)-----
-----
class SentimentGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
num_layers=3):
        """
        -- input_size is the size of the tensor for each word in a
sequence of words.  If you word2vec

```

```

        embedding, the value of this variable will always be
equal to 300.
    -- hidden_size is the size of the hidden state in the RNN
    -- output_size is the size of output of the RNN. For binary
classification of
        input text, output_size is 2.
    -- num_layers creates a stack of GRUs
    """
    super(SentimentGRU, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.gru = nn.GRU(input_size, hidden_size, num_layers,
bidirectional=True)
    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 = out[:, :, :100]
        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_size
        hidden =
weight.new( 6,          1,          self.hidden_size  ).zero_()
        return hidden

## Training loss
criterion = nn.NLLLoss()

## Optimizer
learning_rate = 0.001
model = SentimentGRU(input_size=300, hidden_size=100,
output_size=2).to(device) # 3 output classes
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train model
def train_model(model, dataloader, criterion, optimizer, num_epochs):
    model.train()
    losses = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(dataloader):

```

```

        review_tensor, category, sentiment = data['review'],
data['category'], data['sentiment']
        review_tensor = review_tensor.to(device)
        sentiment = sentiment.to(device)
        optimizer.zero_grad()

        hidden = model.init_hidden().to(device)
        for k in range(review_tensor.shape[1]):
            output, hidden =
model(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0), hidden)
            loss = criterion(output, torch.argmax(sentiment, 1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
        epoch_loss = running_loss / len(dataloader)
        losses.append(epoch_loss)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss}")

# Save generator model and loss plot
torch.save(model.state_dict(),
'./model_100_epochs_extracredit_bidirectional_200.pth')

dictionary_losses = {}

nombre_imagen = 'yes'
dictionary_losses[nombre_imagen] = {
    'criterion1': losses
}

with
open('/home/aolivepe/ECE60146/HW9/loss_100_epochs_extracredit_bidirection
al_200.pkl', 'wb') as archivo:
    pickle.dump(dictionary_losses, archivo)

# Plot training loss
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss over Epochs')
plt.savefig('training_loss_plot.png')
plt.show()

# Evaluate model
def evaluate_model(model, dataloader):
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    neutral_total = 0
    confusion_matrix = torch.zeros(2,2)

```

```

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

        hidden = model.init_hidden().to(device)
        for k in range(review_tensor.shape[1]):
            output, hidden =
model(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 % 50 == 49:
                print("    [i=%4d]    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%%"
% (float(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" % 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)

## Define datasets and dataloaders

```

```
dataset_archive_train =  
"home/aolivepe/ECE60146/HW9/data/sentiment_dataset_train_200.tar.gz"  
  
dataset_archive_test  
= "home/aolivepe/ECE60146/HW9/data/sentiment_dataset_test_200.tar.gz"  
  
path_to_saved_embeddings = "/home/aolivepe/ECE60146/HW9/"  
  
train_dataset =SentimentDataset(  
    train_or_test = 'train',  
    dataset_file = dataset_archive_train,  
    path_to_saved_embeddings =  
path_to_saved_embeddings,  
)  
test_dataset =SentimentDataset(  
    train_or_test = 'test',  
    dataset_file = dataset_archive_test,  
    path_to_saved_embeddings =  
path_to_saved_embeddings,  
)  
  
train_dataloader = DataLoader(train_dataset, batch_size = 1, shuffle =  
True )  
test_dataloader = DataLoader(test_dataset, batch_size = 1, shuffle = True  
)  
  
## Run training and evaluation  
train_model(model, train_dataloader, criterion, optimizer, num_epochs=20)  
evaluate_model(model, test_dataloader)
```