hw8_ChengjunGuo

Chengjun Guo

April 2023

1 GRU

```python
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.project1 = nn.Sequential( nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid() )
        self.project2 = nn.Sequential( nn.Linear( self.input_size + self.hidden_size, self.hidden_size), nn.Tanh() )
        self.project3 = nn.Sequential( nn.Linear( self.hidden_size, self.output_size ), nn.Tanh() )

    def forward(self, x, h, sequence_end=False):
        combined1 = torch.cat((x, h), 2)
        forget_gate = self.project1(combined1)  # rt, zt
        interim = forget_gate * h
        combined2 = torch.cat((x, interim), 2)
        output_interim = self.project2(combined2)   #htilda
        output = (1 - forget_gate) * h + forget_gate * output_interim
        if sequence_end == False:
            return output, output
        else:
            final_out = self.project3(output)
        return final_out, final_out
```

Listing 1: GRU

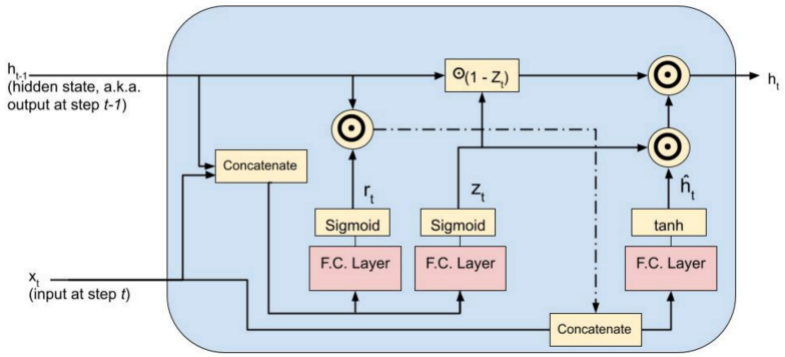

Figure 1: GRU architecture

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1}) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1}) \\
\tilde{h}_t &= tanh(W_h x_t + U_h(r_t \odot h_{t-1})) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

Figure 2: GRU interior formula

The graph is from Prof. kak's lecture. The h on left top is the hidden state output from the last iteration. Correspondingly, the h on right top is the output of hidden state used for next iteration. Inside each iteration, combined1 is the concatenation of hidden state and input vector. Inside the project1, the fully connected layer would convert the size of the combination to the hidden state size. The gate r and z would be processed to between 0 and 1 by sigmoid layer. The first item in the output is the previous hidden state which is multiplied with (1-forget gate). The forget gate decide how much of the previous hidden state will be ignored. The input vector is then concatenated with the forgotten part of the hidden state that is generated as combined2. The project2 would convert the combined2 to hidden state size with a fully connected layer. With the tanh function, it would convert the value h tilde between -1 and 1. The new h tilde added with the remained hidden state would be the new hidden state. If this is the last unit in this layer, the hidden state would be converted to the expected output size with the fully connected layer in the project3. Then the last tanh layer would convert it to [-1,1] as output from GRU. The information retained with the gates would solve the problem of vanishing gradient. The gates would decide the information carried to the next state.

2 Training loss

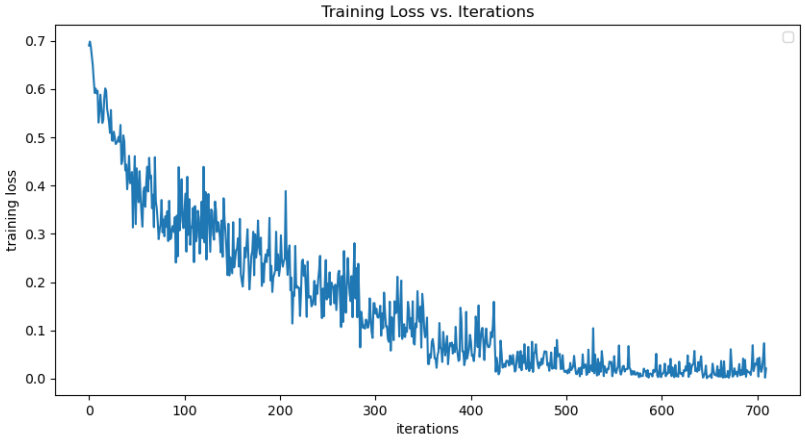

Figure 3: Training Loss with my GRU

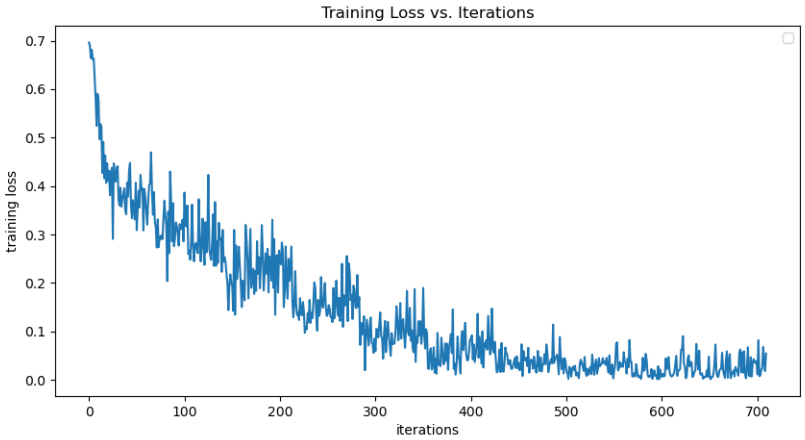

Figure 4: Training Loss with nn GRU

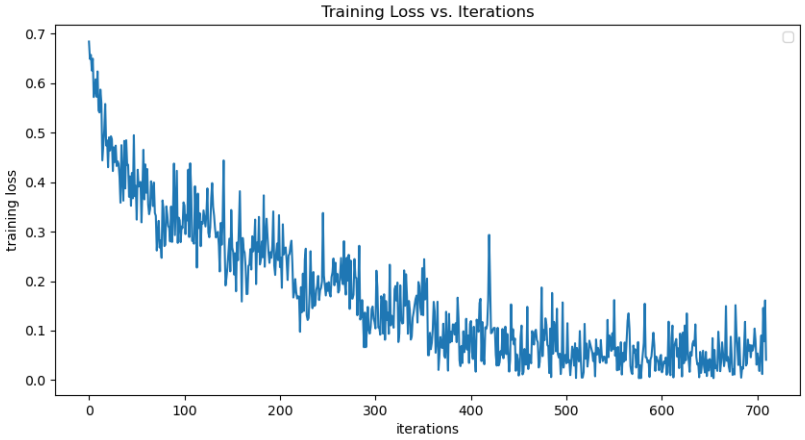

Figure 5: Training Loss with nn GRU bidirectional

3 Confusion matrix

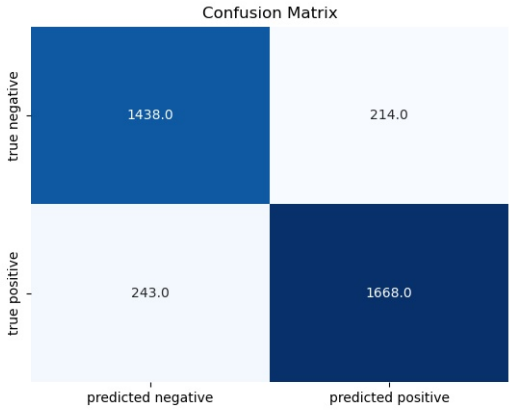

Figure 6: Confusion Matrix with my GRU

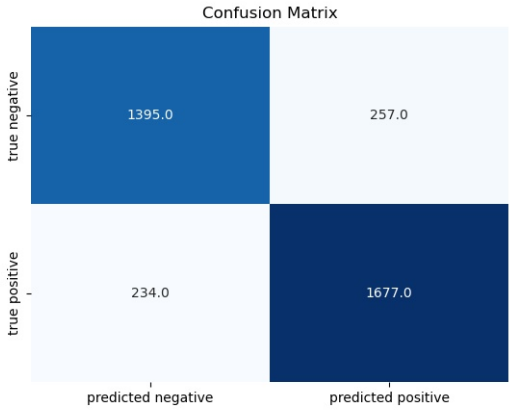

Figure 7: Confusion Matrix with nn GRU

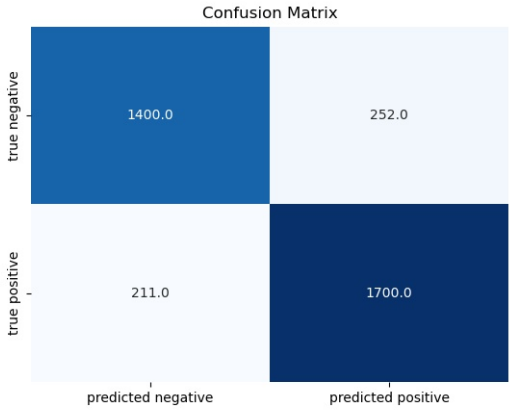

Figure 8: Confusion Matrix with nn GRU bidirectional

4 Accuracy

myGRU:

	predictive negative	predicted positive
true negative	87%	13%
true positive	12.7%	87.3%

nnGRU:

	predictive negative	predicted positive
true negative	84.4%	15.6%
true positive	12.2%	87.8%

bidirectional GRU:

	predictive negative	predicted positive
true negative	84.7%	15.3%
true positive	11%	89.0%

5 Discussion

In the test dataset, my GRU achieves best average performance. The bidirection torch GRU have highest positive prediction. The torch GRU got second high positive performance and lowest performance of negative prediction.

6 Code

```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
import torch
from PIL import Image
from torch.utils.data import DataLoader, Dataset
import copy
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import numpy as np
import gzip
from DLStudio import *
import sys
import random
import pickle
import time
import torch.nn as nn
import gensim
import gensim.downloader as genapi
from gensim.models import KeyedVectors

## reference: https://engineering.purdue.edu/kak/distDLS/DLStudio-2.2.5_CodeOnly.html

class SentimentAnalysisDataset(Dataset):
    """
    In relation to the SentimentAnalysisDataset defined for the TextClassification section of
    DLStudio, the __getitem__() method of the dataloader must now fetch the embeddings from
    the word2vec word vectors.

    Class Path:  DLStudio -> TextClassificationWithEmbeddings -> SentimentAnalysisDataset
    """

    def __init__(self, train_or_test, dataset_file, path_to_saved_embeddings=None):
        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 = './data/'
        f = gzip.open(root_dir + dataset_file, 'rb')
        dataset = f.read()
        if path_to_saved_embeddings is not None:
            if os.path.exists(path_to_saved_embeddings + 'vectors.kv'):
                self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv')
```

```
45            else:
46                print("""\n\nSince this is your first time to
    install the word2vec embeddings, it may take"""
47                      """\na couple of minutes. The embeddings
    occupy around 3.6GB of your disk space.\n\n""")
48                self.word_vectors = genapi.load("word2vec-google-
    news-300")
49                ##  'kv' stands for  "KeyedVectors", a special
    datatype used by gensim because it
50                ##  has a smaller footprint than dict
51                self.word_vectors.save(path_to_saved_embeddings + '
    vectors.kv')
52        if train_or_test == 'train':
53            if sys.version_info[0] == 3:
54                self.positive_reviews_train, self.
    negative_reviews_train, self.vocab = pickle.loads(dataset,
55
                                        encoding='latin1')
56            else:
57                self.positive_reviews_train, self.
    negative_reviews_train, self.vocab = pickle.loads(dataset)
58            self.categories = sorted(list(self.
    positive_reviews_train.keys()))
59            self.category_sizes_train_pos = {category: len(self.
    positive_reviews_train[category]) for category in
60                                             self.categories}
61            self.category_sizes_train_neg = {category: len(self.
    negative_reviews_train[category]) for category in
62                                             self.categories}
63            self.indexed_dataset_train = []
64            for category in self.positive_reviews_train:
65                for review in self.positive_reviews_train[category
    ]:
66                    self.indexed_dataset_train.append([review,
    category, 1])
67            for category in self.negative_reviews_train:
68                for review in self.negative_reviews_train[category
    ]:
69                    self.indexed_dataset_train.append([review,
    category, 0])
70            random.shuffle(self.indexed_dataset_train)
71        elif train_or_test == 'test':
72            if sys.version_info[0] == 3:
73                self.positive_reviews_test, self.
    negative_reviews_test, self.vocab = pickle.loads(dataset,
74
                                       encoding='latin1')
75            else:
76                self.positive_reviews_test, self.
    negative_reviews_test, self.vocab = pickle.loads(dataset)
77            self.vocab = sorted(self.vocab)
78            self.categories = sorted(list(self.
    positive_reviews_test.keys()))
79            self.category_sizes_test_pos = {category: len(self.
    positive_reviews_test[category]) for category in
80                                            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)
        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




class GRUnetWithEmbeddings(nn.Module):
    """
    For this embeddings adapted version of the GRUnet shown earlier, we can assume that
    the 'input_size' for a tensor representing a word is always 300.
    Source: https://blog.floydhub.com/gru-with-pytorch/
    with the only modification that the final output of forward() is now
    routed through LogSoftmax activation.

    Class Path:  DLStudio -> TextClassificationWithEmbeddings -> GRUnetWithEmbeddings
    """

    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        """
        -- 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(GRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_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(1, 1, self.hidden_size).zero_()
        return hidden
```

```
179
180
181 class GRU(nn.Module):
182     def __init__(self, input_size, hidden_size, output_size):
183         super(GRU, self).__init__()
184         self.input_size = input_size
185         self.hidden_size = hidden_size
186         self.output_size = output_size
187         self.project1 = nn.Sequential( nn.Linear(self.input_size +
    self.hidden_size, self.hidden_size), nn.Sigmoid() )
188         self.project2 = nn.Sequential( nn.Linear( self.input_size +
     self.hidden_size, self.hidden_size), nn.Tanh() )
189         self.project3 = nn.Sequential( nn.Linear( self.hidden_size,
     self.output_size ), nn.Tanh() )
190
191     def forward(self, x, h, sequence_end=False):
192         combined1 = torch.cat((x, h), 2)
193         forget_gate = self.project1(combined1)  # rt, zt
194         interim = forget_gate * h
195         combined2 = torch.cat((x, interim), 2)
196         output_interim = self.project2(combined2)   #htilda
197         output = (1 - forget_gate) * h + forget_gate *
    output_interim
198         if sequence_end == False:
199             return output, output
200         else:
201             final_out = self.project3(output)
202         return final_out, final_out
203
204 class myGRUnetWithEmbeddings(nn.Module):
205
206     def __init__(self, input_size, hidden_size, output_size,
    num_layers=1):
207         super(myGRUnetWithEmbeddings, self).__init__()
208         self.input_size = input_size
209         self.hidden_size = hidden_size
210         self.num_layers = num_layers
211         self.gru = GRU(input_size, hidden_size, output_size)
212         self.fc = nn.Linear(hidden_size, output_size)
213         self.relu = nn.ReLU()
214         self.logsoftmax = nn.LogSoftmax(dim=1)
215
216     def forward(self, x, h):
217         out, h = self.gru(x, h)
218         out = self.fc(self.relu(out[:, -1]))
219         out = self.logsoftmax(out)
220         return out, h
221
222     def init_hidden(self):
223         weight = next(self.parameters()).data
224         #                   num_layers  batch_size    hidden_size
225         hidden = weight.new(1, 1, self.hidden_size).zero_()
226         return hidden
227
228 class BidirGRUnetWithEmbeddings(nn.Module):
229
```

```
    def __init__(self, input_size, hidden_size, output_size,
    num_layers=1):
        super(BidirGRUnetWithEmbeddings, 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, output_size,
    bidirectional=True)
        self.fc = nn.Linear(hidden_size * 2, output_size)   #
    bidirectional
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.relu(out[:, -1])
        out = self.fc(out)
        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(4, 1, self.hidden_size).zero_()
        return hidden

def run_code_for_training_for_text_classification_with_GRU_word2vec
    (net, train_dataloader,path_saved_model):
    epochs = 10
    filename_for_out = "performance_numbers_" + str(epochs) + ".txt
    "
    FILE = open(filename_for_out, 'w')
    net = copy.deepcopy(net)
    net = net.to(device)
    ##  Note that the GRUnet now produces the LogSoftmax output:
    criterion = torch.nn.NLLLoss()
    accum_times = []
    optimizer = torch.optim.Adam(net.parameters(),lr=1e-3,betas
    =(0.9, 0.99), eps = 1e-4)    #default eps might get training
    loss to nan
    training_loss_tally = []
    start_time = time.perf_counter()
    for epoch in range(epochs):
        print("")
        running_loss = 0.0
        for i, data in enumerate(train_dataloader):
            review_tensor,category,sentiment = data['review'], data
    ['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            ## The following type conversion needed for MSELoss:
            ##sentiment = sentiment.float()
            optimizer.zero_grad()
            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)
```

```
            loss = criterion(output, torch.argmax(sentiment, 1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
            if i % 200 == 199:
                avg_loss = running_loss / float(200)
                training_loss_tally.append(avg_loss)
                current_time = time.perf_counter()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time:%4d secs]
        loss: %.5f" % (epoch+1,i+1, time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                FILE.write("%.5f\n" % avg_loss)
                FILE.flush()
                running_loss = 0.0
    torch.save(net.state_dict(), path_saved_model)
    print("Total Training Time: {}".format(str(accum_times[-1])))
    print("\nFinished Training\n\n")
    return training_loss_tally


def run_code_for_testing_text_classification_with_GRU_word2vec(net,
    test_dataloader,path_saved_model):
    net.load_state_dict(torch.load(path_saved_model))
    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['sentiment']
            hidden = net.init_hidden()
            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" % (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)
    return confusion_matrix



if __name__ == '__main__':
    if torch.cuda.is_available() == True:
        device = torch.device("cuda:0")
    else:
        device = torch.device("cpu")
    # mygru
    # dataset_archive_train = "sentiment_dataset_train_400.tar.gz"
    # net = myGRUnetWithEmbeddings(input_size=300, hidden_size=100,
    output_size=2)
    # train_dataset = SentimentAnalysisDataset("train",
    dataset_archive_train,"data/word2vec/")
    # train_dataloader = torch.utils.data.DataLoader(train_dataset,
     batch_size=1, shuffle=True, num_workers=2)
    # loss =
    run_code_for_training_for_text_classification_with_GRU_word2vec
    (net, train_dataloader,'net1.pth')
    # plt.figure(figsize=(10, 5))
    # plt.title("Training Loss vs. Iterations")
    # plt.plot(loss)
    # plt.xlabel("iterations")
    # plt.ylabel("training loss")
    # plt.legend()
    # plt.savefig("training_loss.png")
    # plt.show()

    # dataset_archive_test = "sentiment_dataset_test_400.tar.gz"
    # net = myGRUnetWithEmbeddings(input_size=300, hidden_size=100,
    output_size=2)
    # path_saved_model = 'net1.pth'
    # test_dataset = SentimentAnalysisDataset("test",
    dataset_archive_test,"data/word2vec/")
    # test_dataloader = torch.utils.data.DataLoader(test_dataset,
    batch_size=1, shuffle=True, num_workers=2)
    # cm =
    run_code_for_testing_text_classification_with_GRU_word2vec(net,
     test_dataloader,path_saved_model)
    # plt.figure()
```

```
    # sns.heatmap(cm,annot = True, fmt = "", cmap = "Blues", cbar = False, xticklabels = ['predicted negative', 'predicted positive'], yticklabels = ['true negative', 'true positive'])
    # plt.title("Confusion Matrix")
    # plt.savefig("cm1.jpg")

    # nn gru
    # dataset_archive_train = "sentiment_dataset_train_400.tar.gz"
    # net = GRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2)
    # train_dataset = SentimentAnalysisDataset("train", dataset_archive_train,"data/word2vec/")
    # train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=1, shuffle=True, num_workers=2)
    # loss = run_code_for_training_for_text_classification_with_GRU_word2vec(net, train_dataloader,'net2.pth')
    # plt.figure(figsize=(10, 5))
    # plt.title("Training Loss vs. Iterations")
    # plt.plot(loss)
    # plt.xlabel("iterations")
    # plt.ylabel("training loss")
    # plt.legend()
    # plt.savefig("training_loss_2.png")
    #
    # dataset_archive_test = "sentiment_dataset_test_400.tar.gz"
    # net = GRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2)
    # path_saved_model = 'net2.pth'
    # test_dataset = SentimentAnalysisDataset("test", dataset_archive_test,"data/word2vec/")
    # test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=1, shuffle=True, num_workers=2)
    # cm = run_code_for_testing_text_classification_with_GRU_word2vec(net, test_dataloader,path_saved_model)
    # plt.figure()
    # sns.heatmap(cm,annot = True, fmt = "", cmap = "Blues", cbar = False, xticklabels = ['predicted negative', 'predicted positive'], yticklabels = ['true negative', 'true positive'])
    # plt.title("Confusion Matrix")
    # plt.savefig("cm2.jpg")

    # nn bidirectional gru
    dataset_archive_train = "sentiment_dataset_train_400.tar.gz"
    net = BidirGRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2)
    train_dataset = SentimentAnalysisDataset("train", dataset_archive_train,"data/word2vec/")
    train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=1, shuffle=True, num_workers=2)
    loss = run_code_for_training_for_text_classification_with_GRU_word2vec(net, train_dataloader,'net3.pth')
    plt.figure(figsize=(10, 5))
    plt.title("Training Loss vs. Iterations")
    plt.plot(loss)
```

```
    plt.xlabel("iterations")
    plt.ylabel("training loss")
    plt.legend()
    plt.savefig("training_loss_3.png")
    #
    dataset_archive_test = "sentiment_dataset_test_400.tar.gz"
    net = BidirGRUnetWithEmbeddings(input_size=300, hidden_size
    =100,output_size=2)
    path_saved_model = 'net3.pth'
    test_dataset = SentimentAnalysisDataset("test",
    dataset_archive_test,"data/word2vec/")
    test_dataloader = torch.utils.data.DataLoader(test_dataset,
    batch_size=1, shuffle=True, num_workers=2)
    cm = run_code_for_testing_text_classification_with_GRU_word2vec
    (net, test_dataloader,path_saved_model)
    plt.figure()
    sns.heatmap(cm,annot = True, fmt = "", cmap = "Blues", cbar =
    False, xticklabels = ['predicted negative', 'predicted positive
    '], yticklabels = ['true negative', 'true positive'])
    plt.title("Confusion Matrix")
    plt.savefig("cm3.jpg")
```