SPRING 2023 ECE 60146 – Homework 8

Sahithi Kodali – 34789866
kodali1@purdue.edu

3.1 Sentiment Analysis with own GRU

To implement the own GRU network a GRU module is implemented using the main equations of GRU logic on Slide 59 of Prof. Kak's week – 12 lecture as below.

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

Similar to the equations an update gate and reset gate are initialized whose main role is to control the amount of information to retain for the input (update function z) and how much to forget (reset function r) that is not relevant to the context for the input generated. A candidate hidden state is dependent on the current value of input sequence and the previous value of the hidden state. This ensures that the current input can be combined with previous contextual information. Further using this candidate hidden state and previous hidden state along with the update gate, the hidden state for the current update is computed. The own implementation is an exact computation of the equations above as seen in Figure 1.

This GRU is wrapped and used in RNN network where the hidden state is computed and further used to compute the next output. A logsoftmax function is used to attain shape of output that can be compared with sentiment label during loss calculation as shown in Figure 2.

We can say that gating mechanism can successfully reduce the vanishing gradient problem by forgetting/resetting the information that is not relevant/context based for generating the next output in sequence. This way only the necessary dependencies are retained while the not very relevant dependencies are reset leading to less learnable parameters and thus reduction in vanishing gradients.

```
#Own GRU module
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        # update gate parameters
        self.Wz = nn.Linear(input_size, hidden_size)
        self.Uz = nn.Linear(hidden_size, hidden_size)
        # reset gate parameters
        self.Wr = nn.Linear(input_size, hidden_size)
        self.Ur = nn.Linear(hidden_size, hidden_size)
        # candidate hidden state parameters
        self.Wh = nn.Linear(input_size, hidden_size)
        self.Uh = nn.Linear(hidden_size, hidden_size)
```

```
    def forward(self, x, h):
        # compute update gate and reset cate
        z = torch.sigmoid(self.Wz(x) + self.Uz(h))
        r = torch.sigmoid(self.Wr(x) + self.Ur(h))
        # compute candidate hidden state
        h_tilda = torch.tanh(self.Wh(x) + self.Uh(r * h))
        # compute new hidden state
        h_new = (1 - z) * h + z * h_tilda
        return h_new

#implemented similar to the basic RNN network (from net sources and Prof Kak' codes)
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.gru = GRU(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        h = torch.zeros(1, self.hidden_size) # initialize hidden state with zeros
        for i in range(x.shape[0]):
            h = self.gru(x[i], h)
        out = self.fc(h)
        out = self.logsoftmax(out)
        return out
```

Figure 1: Own GRU code block

The plot of loss vs iterations for the own GRU implementation can be seen in Figure 2 and the results attained can be seen in Figure 3.

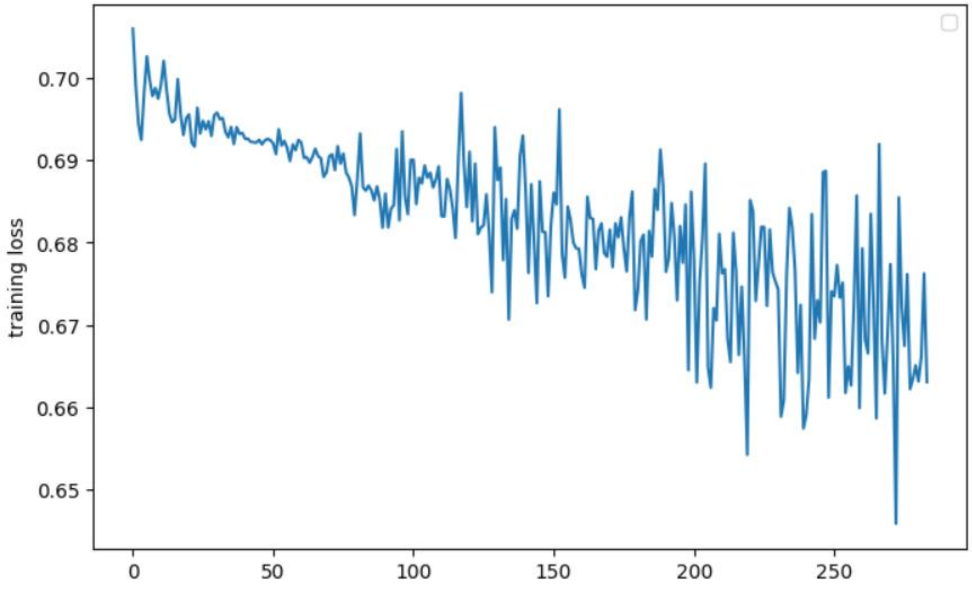

Figure 2: Plot of loss vs iterations for the own GRU

```
Confusion matrix:
[[ 468 1184]
 [ 164 1747]]
Accuracy of the network on the test data: 62 %
```

Figure 3: Confusion matrix and accuracy attained using own GRU

3.2 Sentiment Analysis using torch.nn.GRU

The other two RNN networks implemented are using nn.torch.GRU, one of which is implemented by giving the hyperparameter 'bidirectional = True'. The plots of loss vs iterations for these two networks can be seen in Figure 4 and Figure 6, while the quantitative results can be seen in Figure 5 and Figure 7.

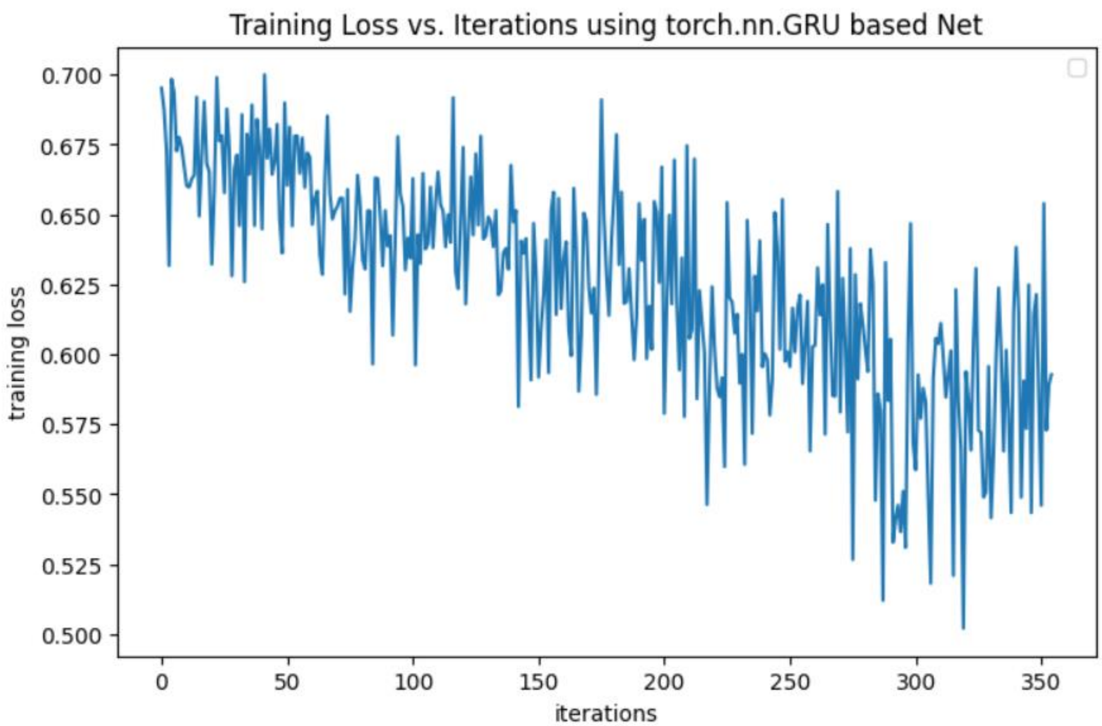

Figure 4: Plot of loss vs iterations for torch.nn.GRU network

```
Confusion matrix using nn.GRU based network:
[[1159  493]
 [ 626 1285]]
Accuracy of the nn.GRU based network on the test data: 68 %
```

Figure 5: Confusion matrix and accuracy attained using torch.nn.GRU network

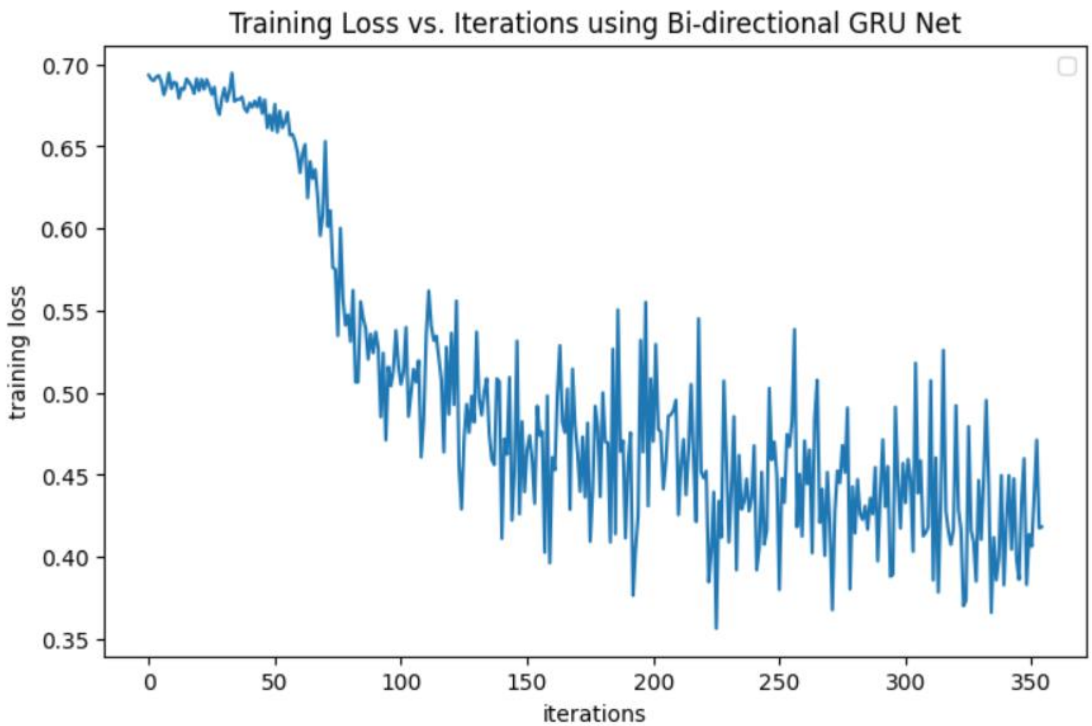

Figure 6: Plot of loss vs iterations for bidirectional torch.nn.GRU network

```
Confusion matrix of b-directional GRU:
[[1339  313]
 [ 442 1469]]
Accuracy of the bi-directional GRU network on the test data: 78 %
```

Figure 7: Confusion matrix and accuracy attained using bidirectional torch.nn.GRU network

<u>Comparing the three GRUs implemented above:</u>

We can observe that the bi-directional GRU attained the maximum accuracy of 78% among the three RNN networks implemented, with the second-best accuracy attained as 68% by non-bidirectional torch.nn.GRU based network. We can also observe a curved loss decrease in the training of bi-directional network while the others had steady increase/decrease in reducing loss. An interesting observation is the vanilla GRU which attained accuracy of 62% predicted more true negatives than the bidirectional GRU however, bi-directional GRU stood in better position comparatively. We can say that the bi-directional GRU domination in performance can be due to the fact of considering both the past and future contexts i.e., both directions of sequence context being considered in prediction leading to better predictions. All these GRUs' performance can still be improved by using much deeper and advanced configuration of networks.

SOURCE CODE:

```
# -*- coding: utf-8 -*-
"""HW8.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ZnkwP4s8OqlxceI_m-Fg_xYz58fHhOvn
"""

# Commented out IPython magic to ensure Python compatibility.
#import libraries required

import gzip
import pickle
# %matplotlib inline
from pycocotools.coco import COCO
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
import random
import os
import zipfile
from shutil import copyfile
import sys
import copy

import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import pandas as pd
import torchvision
import torchvision.transforms as tvt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

#give paths
path_to_saved_embeddings = '/content/drive/MyDrive/Purdue/HW8/'
root_dir = '/content/drive/MyDrive/Purdue/HW8/data/'

dataset_file_train = 'sentiment_dataset_train_400.tar.gz'
dataset_file_test = 'sentiment_dataset_test_400.tar.gz'
```

```
#create dataset and dataloader
class SADataset(Dataset):

    #intializations
    def __init__(self, root_dir, dataset_file, train_or_test,
path_to_saved_embeddings = None):
        super().__init__()
        self.path_to_saved_embeddings = path_to_saved_embeddings
        self.train_or_test = train_or_test
        self.root_dir = 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 )
        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

train_data_400 = SADataset(root_dir, dataset_file_train, 'train',
path_to_saved_embeddings)
test_data_400 = SADataset(root_dir, dataset_file_test, 'test',
path_to_saved_embeddings)

len(test_data_400)

# Set device to GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

batch_size = 1
num_workers = 2
```

```python
#create dataloader
train_data_loader = DataLoader(train_data_400, batch_size = batch_size, shuffle =
True, num_workers=num_workers)
test_data_loader = DataLoader(test_data_400, batch_size = batch_size, shuffle =
True, num_workers=num_workers)

data = next(iter(test_data_loader))
print(data['review'].shape)

count = 0
for i, data in enumerate(train_data_loader):
    review_tensor,category,sentiment = data['review'], data['category'],
data['sentiment']
    count += 1
print(count)

#Own GRU module
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        # update gate parameters
        self.Wz = nn.Linear(input_size, hidden_size)
        self.Uz = nn.Linear(hidden_size, hidden_size)
        # reset gate parameters
        self.Wr = nn.Linear(input_size, hidden_size)
        self.Ur = nn.Linear(hidden_size, hidden_size)
        # candidate hidden state parameters
        self.Wh = nn.Linear(input_size, hidden_size)
        self.Uh = nn.Linear(hidden_size, hidden_size)

    def forward(self, x, h):
        # compute update gate and reset cate
        z = torch.sigmoid(self.Wz(x) + self.Uz(h))
        r = torch.sigmoid(self.Wr(x) + self.Ur(h))
        # compute candidate hidden state
        h_tilda = torch.tanh(self.Wh(x) + self.Uh(r * h))
        # compute new hidden state
        h_new = (1 - z) * h + z * h_tilda
        return h_new

#implemented similar to the basic RNN network (from net sources and Prof Kak'
codes)
```

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.gru = GRU(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        h = torch.zeros(1, self.hidden_size) # initialize hidden state with zeros
        for i in range(x.shape[0]):
            h = self.gru(x[i], h)
        out = self.fc(h)
        out = self.logsoftmax(out)
        return out

#training

hidden_size = 100

net2 = RNN(input_size = 300, hidden_size=128, output_size=2)
net2 = copy.deepcopy(net2)
net2 = net2.to(device)

net2.train()

##  Note that the GRUnet now produces the LogSoftmax output:
criterion = nn.NLLLoss()
optimizer = optim.Adam(net2.parameters(), lr=1e-3, betas=(0.9, 0.999))
training_loss_tally_own = []


for epoch in range(epochs):
    print("")
    running_loss = 0.0

    for i, data in enumerate(train_data_loader,0):
        review_tensor,category,sentiment = data['review'], data['category'],
data['sentiment']
        review_tensor = review_tensor.to(device)
        sentiment = sentiment.to(device)

        optimizer.zero_grad()

        output = net2(review_tensor)
```

```
        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_own.append(avg_loss)

            print("[epoch:%d  iter:%4d]     loss: %.5f" % (epoch+1,i+1,avg_loss))
            running_loss = 0.0

print("\nFinished Training\n\n")

#plotting
plt.figure(figsize=(8,5))
plt.title("Training Loss vs. Iterations")
plt.plot(training_loss_tally_own)

plt.xlabel("iterations")
plt.ylabel("training loss")
plt.legend()
plt.savefig("training_loss.png")
plt.show()

#testing
from sklearn.metrics import confusion_matrix

# Set the network to evaluation mode
net2.eval()

# Initialize variables for the confusion matrix
all_predictions = []
all_targets = []

# Iterate over the test data and predict the sentiment for each review
with torch.no_grad():
    for data in test_data_loader:
        review_tensor, category, sentiment = data['review'], data['category'],
data['sentiment']
        review_tensor = review_tensor.to(device)
        sentiment = sentiment.to(device)  # Move sentiment tensor to GPU

        output, hidden = net2(review_tensor)
```

```
        _, predicted = torch.max(output.data, 1)
        all_predictions.extend(predicted.cpu().numpy())
        all_targets.extend(torch.argmax(sentiment, 1).cpu().numpy())

# Compute the confusion matrix
cm = confusion_matrix(all_targets, all_predictions)

# Print the confusion matrix
print("Confusion matrix:")
print(cm)

# Compute the overall accuracy
overall_accuracy = 100 * cm.trace() / cm.sum()
print('Accuracy of the network on the test data: %d %%' % overall_accuracy)

#implementation using torch.nn.GRU
class GRU_Net(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, batch_size):
        weight = next(self.parameters()).data
        hidden = weight.new(self.num_layers, batch_size, 
self.hidden_size).zero_()
        return hidden

#training
epochs = 5


filename_for_out = "performance_numbers_" + str(epochs) + ".txt"
```

```
FILE = open(filename_for_out, 'w')

net1 = GRU_Net(input_size=300, hidden_size=100, output_size=2, num_layers=2)
net1 = copy.deepcopy(net1)
net1 = net1.to(device)

net1.train()

##  Note that the GRUnet now produces the LogSoftmax output:
criterion = nn.NLLLoss()
optimizer = optim.Adam(net1.parameters(), lr=1e-3, betas=(0.9, 0.999))
training_loss_tally_GRU_INI = []

for epoch in range(epochs):
    print("")
    running_loss = 0.0

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

        optimizer.zero_grad()
        hidden = net1.init_hidden(review_tensor.shape[1]).to(device)

        output, hidden = net1(review_tensor, 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_GRU_INI.append(avg_loss)

            print("[epoch:%d  iter:%4d]     loss: %.5f" % (epoch+1,i+1,avg_loss))

            FILE.write("%.5f\n" % avg_loss)
            FILE.flush()
            running_loss = 0.0

print("\nFinished Training\n\n")
```

```
#plotting
plt.figure(figsize=(8,5))
plt.title("Training Loss vs. Iterations using torch.nn.GRU based Net")
plt.plot(training_loss_tally_GRU_INI)

plt.xlabel("iterations")
plt.ylabel("training loss")
plt.legend()
plt.savefig("training_loss_GRU.png")
plt.show()

#Evaluation
from sklearn.metrics import confusion_matrix

# Set the network to evaluation mode
net1.eval()

# Initialize variables for the confusion matrix
all_predictions = []
all_targets = []

# Iterate over the test data and predict the sentiment for each review
with torch.no_grad():
    for data in test_data_loader:
        review_tensor, category, sentiment = data['review'], data['category'],
data['sentiment']
        review_tensor = review_tensor.to(device)
        sentiment = sentiment.to(device)

        hidden = net1.init_hidden(batch_size=1).to(device)
        list_of_outputs = []
        for k in range(review_tensor.shape[1]):
            output, hidden =
net1(torch.unsqueeze(torch.unsqueeze(review_tensor[0, k], 0), 0), hidden)
            list_of_outputs.append(output)
        output = list_of_outputs[-1]

        _, predicted = torch.max(output.data, 1)
        all_predictions.extend(predicted.cpu().numpy())
        all_targets.extend(torch.argmax(sentiment, 1).cpu().numpy())

# Compute the confusion matrix
cm = confusion_matrix(all_targets, all_predictions)

# Print the confusion matrix
```

```
print("Confusion matrix using nn.GRU based network:")
print(cm)

# Compute the overall accuracy
overall_accuracy = 100 * cm.trace() / cm.sum()
print('Accuracy of the nn.GRU based network on the test data: %d %%' %
overall_accuracy)

#Implementation of Bi-directional GRU
class GRU_Net_Bi(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers,
drop_prob=0.2, bidirectional=True):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = bidirectional
        self.num_directions = 2 if bidirectional else 1
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True,
bidirectional=bidirectional)
        self.fc = nn.Linear(hidden_size*self.num_directions, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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

#training
epochs = 5

filename_for_out = "performance_numbers_" + str(epochs) + ".txt"
FILE = open(filename_for_out, 'w')
```

```
net = GRU_Net_Bi(input_size=300, hidden_size=100, output_size=2, num_layers=2,
bidirectional=True)
net = copy.deepcopy(net)
net = net.to(device)

net.train()

##  Note that the GRUnet now produces the LogSoftmax output:
criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr=1e-5, betas=(0.9, 0.999))
training_loss_tally_GRU = []

for epoch in range(epochs):
    print("")
    running_loss = 0.0

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

        optimizer.zero_grad()
        hidden = net.init_hidden(review_tensor.shape[0]).to(device)

        output, hidden = net(review_tensor, hidden)
        print(output.shape)

        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_GRU.append(avg_loss)

            print("[epoch:%d  iter:%4d]     loss: %.5f" % (epoch+1,i+1,avg_loss))

            FILE.write("%.5f\n" % avg_loss)
            FILE.flush()
            running_loss = 0.0

print("\nFinished Training\n\n")
```

```
#plotting
plt.figure(figsize=(8,5))
plt.title("Training Loss vs. Iterations using Bi-directional GRU Net")
plt.plot(training_loss_tally_GRU)

plt.xlabel("iterations")
plt.ylabel("training loss")
plt.legend()
plt.savefig("training_loss_GRU_BiDirectional.png")
plt.show()

#testing
from sklearn.metrics import confusion_matrix

# Set the network to evaluation mode
net.eval()

# Initialize variables for the confusion matrix
all_predictions = []
all_targets = []

# Iterate over the test data and predict the sentiment for each review
with torch.no_grad():
    for data in test_data_loader:
        review_tensor, category, sentiment = data['review'], data['category'],
data['sentiment']
        review_tensor = review_tensor.to(device)
        sentiment = sentiment.to(device)  # Move sentiment tensor to GPU

        hidden = net.init_hidden(batch_size=1).to(device)
        list_of_outputs = []
        for k in range(review_tensor.shape[1]):
            output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,
k], 0), 0), hidden)
            list_of_outputs.append(output)
        output = list_of_outputs[-1]

        _, predicted = torch.max(output.data, 1)
        all_predictions.extend(predicted.cpu().numpy())
        all_targets.extend(torch.argmax(sentiment, 1).cpu().numpy())

# Compute the confusion matrix
cm = confusion_matrix(all_targets, all_predictions)

# Print the confusion matrix
```

```
print("Confusion matrix of b-directional GRU:")
print(cm)

# Compute the overall accuracy
overall_accuracy = 100 * cm.trace() / cm.sum()
print('Accuracy of the bi-directional GRU network on the test data: %d %%' %
overall_accuracy)
```

******