Homework 8 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

April 17, 2023

1 Introduction

In this homework, the programming tasks give an overview of using a Recurrent Neural Network for text classification and sentiment analysis. It introduces the idea of Gated Recurrent Units or GRUs which are widely employed for extracting meaningful time-evolving text features through long chains of feedback for predicting the next word based on the context of previously occurring words. Moreover, it experiments with the usage of word embeddings called *word2vec* on top of the GRU models to deal with different vocab sizes of the text dataset. In this assignment, a Recurrent Neural Network is created consisting of a custom implementation of the GRU cell to classify positive and negative review texts. The networks were trained and validated using a subset of the Amazon Reviews dataset provided by *DLStudio*. Finally, the performance of the networks was compared by computing the confusion matrix to understand which reviews were correctly or incorrectly labeled as positive or negative. Some ideas related to the creation of the Sentiment Analysis dataset and the training and validation routine were taken from the source code of the **DLStudio** module and from the previous year's solutions for completing this assignment. Also, the idea for the implementation of the custom GRU model was taken from this repository and modified accordingly for this assignment.

2 Methodology

Initially, to get familiar with the code for training and testing recurrent neural networks with GRU, the scripts *text_classification_with_GRU_word2vec.py* & *power_load_prediction_with_pmGRU.py* from the **Examples** directory in the **DLStudio** module were run. These scripts illustrated how the neural networks were constructed with GRU cells along with word embeddings and gave an idea of the performance difference of different GRU variants. Using a similar approach for the programming tasks, a custom GRU class was created and used in the *CustomGRUnetWithEmbeddings* network for classifying positive and negative reviews. A training routine was created to train and log the losses at periodic checkpoints. For evaluation, a method was created to calculate the confusion matrix to measure the accuracy of the models and how the models performed in classifying each of the reviews. The following section gives a detailed description of each of the approaches and the results obtained from the experiments performed on the different GRU models.

3 Implementation and Results

3.1 Task 3: Programming Tasks

3.1.1 Sentiment Analysis using Custom GRU

- **Sentiment Analysis Dataset**: A subset of the Amazon Reviews dataset was used for this homework. The training and testing data zip files called *sentiment_dataset_train_400.tar.gz* and *sentiment_dataset_test_400.tar.gz* respectively were downloaded from the DLStudio website. It contained a total of 14227 reviews in the training set and 3563 reviews in the validation set.

  A custom *SentimentAnalysisDataset* class was created from *torch.utils.data.Dataset* class to load the review texts from each of the corresponding zip files. Also, the pre-trained *word2vec* embeddings called *word2vec-google-news-300* were used for mapping the words in the reviews to 300-length embedding vectors. Finally, a *Dataloader* was created to wrap the dataset for processing the text embeddings in single batches for training and validation.

- **Custom GRU**: In this task, a custom GRU cell network *GRUCell* was created along a *customGRU* network. The GRUCell consists of reset gate, update gate, and the network for computing the candidate hidden states. A series of GRUCells are then used in sequence to form the *customGRU*. The following equations provide the calculations required for getting the output for each of these gates.

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

  Here, x_t and h_{t-1} are the input vector and the hidden state, respectively, at the iteration index t. The output is the hidden state h_t to be used for subsequent iterations. σ is the sigmoid activation function and *tanh* is the *tanH* activation. z_t and r_t are the update and the reset gate outputs at the iteration index t. $\tilde{h_t}$ is the candidate hidden state at the current index t. Linear layers followed by the corresponding activation functions were used to compute the output of the gates in the GRUCells. Finally, in the *customGRU* model the text is processed word by word through the sequence of GRU cells, and the final layer output and the hidden states are returned. The following code block gives a description of the *GRUCell* and *customGRU* network.

```
# GRU Cell
from torch.autograd import Variable

class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size, bias=False):
        super(GRUCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias

        self.input2hidden = nn.Linear(input_size, 3 * hidden_size, bias=bias) # [W_z, W_r, W_h]
        self.hidden2hidden = nn.Linear(hidden_size, 3 * hidden_size, bias=bias) # [U_z, U_r, U_h]

        self.reset_parameters()

    def reset_parameters(self):
        std = 1.0 / np.sqrt(self.hidden_size)
        for w in self.parameters():
            w.data.uniform_(-std, std)

    def forward(self, input, hx=None):
        if hx is None:
```

```
            hx = Variable(input.new_zeros(input.size(0), self.hidden_size))

        x_t = self.input2hidden(input)
        h_t = self.hidden2hidden(hx)

        x_reset, x_update, x_new = x_t.chunk(3, 1)
        h_reset, h_update, h_new = h_t.chunk(3, 1)

        reset_gate = torch.sigmoid(x_reset + h_reset) #reset gate
        update_gate = torch.sigmoid(x_update + h_update) #update gate
        new_gate = torch.tanh(x_new + (reset_gate * h_new)) #candidate hidden state

        hy = update_gate * hx + (1 - update_gate) * new_gate

        return hy


# Custom GRU
class CustomGRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, bias = False):
        super(CustomGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bias = bias

        self.cell_list = nn.ModuleList()
        self.cell_list.append(GRUCell(self.input_size, self.hidden_size, self.bias))
        for l in range(1, self.num_layers):
            self.cell_list.append(GRUCell(self.hidden_size, self.hidden_size, self.bias))

    def forward(self, input, hx=None):
        if hx is None:
            if torch.cuda.is_available():
                h0 = Variable(torch.zeros(self.num_layers, input.size(1), self.hidden_size).cuda())
            else:
                h0 = Variable(torch.zeros(self.num_layers, input.size(1), self.hidden_size))
        else:
             h0 = hx

        outs = []
        hidden = list()

        for layer in range(self.num_layers):
            hidden.append(h0[layer, :, :])

        for t in range(input.size(0)):
            for layer in range(self.num_layers):
                if layer == 0:
                    hidden_out = self.cell_list[layer](input[t, :, :], hidden[layer])
                else:
                    hidden_out = self.cell_list[layer](hidden[layer - 1], hidden[layer])
                hidden[layer] = hidden_out

            outs.append(hidden_out)

        return outs, hidden
```

An inactive update gate means that the information in the previous iteration is propagated to the current state. So, the information travels through the time steps which provides a long-term dependency. An active reset gate means the candidate hidden state($\tilde{h_t}$) has information integrated into it from the previous time step. At the same time, if the update gate(z_t) is active, this information will be conveyed to the next steps but will die out quickly. This is a short-term dependency. By incorporating both these dependencies, the network can model a long chain of evolving states and hence can solve the problem of vanishing gradients.

- **Custom GRUnetWithEmbeddings architecture**: The *CustomGRUnetWithEmbeddings* model was created as per the *GRUnetWithEmbeddings* model given in the *DLStudio*. The previously

defined *customGRU* units were used instead of the *torch.nn.GRU* units. For the classification network, the last output of the *customGRU* network is passed through ReLU layers followed by a fully-connected layer. The log of the softmax of the predicted values is then calculated to compute the NLL Loss. The dimension of the hidden state was set as 100 and the number of hidden layers as 2. The total number of learnable layers of the entire network came out to be **6** and the total number of learnable parameters was ∼**1.8M**. The following code snippet shows the *CustomGRUnetWithEmbeddings* network.

```
# Custom GRUnetWithEmbeddings
class CustomGRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(CustomGRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = CustomGRU(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(  2,          1,         self.hidden_size    ).zero_()
        return hidden
```

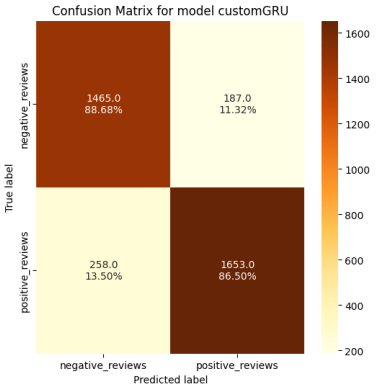
- **Training and Validating custom GRU**: A training routine was created to get the embedding vectors of the reviews from the dataloader and compute the NLL loss based on the class prediction of the custom network. The average losses were also plotted at periodic intervals. A validation routine was also created in which based on the class predictions, a confusion matrix was generated and the overall percentage accuracy was calculated.

  Using the training routine created, the network was trained for 5 epochs optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.001. In order to validate the network, a validation routine was created where the predicted labels were obtained via. model evaluation. These predicted labels were then used to calculate the confusion matrix. Fig. 1 shows the training loss and the confusion matrix for *CustomGRUnetWithEmbeddings*. The classification accuracy was about **87.54%**.

3.1.2 Sentiment Analysis using torch.nn GRU

- **torch.nn GRU** - Here, the *customGRU* network was replaced with the *torch.nn.GRU* implementation to form the *GRUnetWithEmbeddings* model. The total number of learnable layers of the entire network came out to be **10** and the total number of learnable parameters was ∼**1.81M**. Using the training routine created, the network was trained for 5 epochs with optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.001. In order to validate the network, the previously created validation routine was used where the predicted labels were obtained via. model evaluation. These predicted labels were then used to calculate the confusion matrix. Fig. 2 shows the training loss and the confusion matrix for *GRUnetWithEmbeddings*. The overall classification accuracy was about **87.68%**.

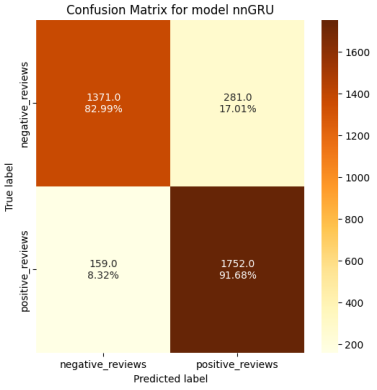
(a) Training loss vs Iterations for *CustomGRUnetWithEmbeddings*



(b) Confusion Matrix for *CustomGRUnetWithEmbeddings* validations. Accuracy = 87.54%

Figure 1: Training loss and the confusion matrix for *CustomGRUnetWithEmbeddings*

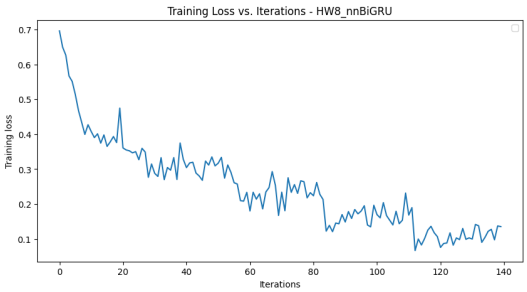
(a) Training loss vs Iterations for *GRUnetWithEmbeddings*



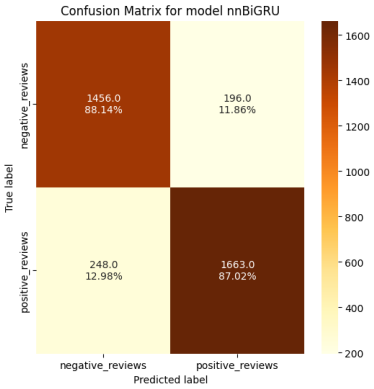
(b) Confusion Matrix for *GRUnetWithEmbeddings* validations. Accuracy = 87.68%

Figure 2: Training loss and the confusion matrix for *GRUnetWithEmbeddings*

- **Bidirectional GRU** - Here, the parameter *bidirectional* was set as true in the *torch.nn.GRU* network to form the *BiGRUnetWithEmbeddings* model. The corresponding output and hidden state dimensions were adjusted for this bidirectional case. The total number of learnable layers of the entire network came out to be **18** and the total number of learnable parameters was ∼**4.22M**. Using the same training routine earlier, the network was trained for 5 epochs with optimizer as Adam having parameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$ and learning rate 0.001. In order to validate the network, the previously created validation routine was used where the predicted labels were obtained via. model evaluation. These predicted labels were then used to calculate the confusion matrix. Fig. 3 shows the training loss and the confusion matrix for *BiGRUnetWithEmbeddings*. The overall classification accuracy was about **87.56%**.
- **Results & Comparison**: The validation accuracies of the *CustomGRUnetWithEmbeddings*, *GRUnetWithEmbeddings* and *BiGRUnetWithEmbeddings* networks were 87.54%, 87.68% and



(a) Training loss vs Iterations for *BiGRUnetWithEmbeddings*



(b) Confusion Matrix for *BiGRUnetWithEmbeddings* validations. Accuracy = 87.56%

Figure 3: Training loss and the confusion matrix for *BiGRUnetWithEmbeddings*

87.56% respectively. It can be seen that the classification accuracies are quite similar in all the cases. The bidirectional GRU performed almost similarly to the unidirectional GRU. This is because the bidirectional GRUs doesn't provide any additional advantage by incorporating the context of future words in terms of the positivity or negativity of the reviews. Bidirectional GRUs are more effective in machine translation where the syntax of the language may depend on future context. However, the custom GRU implementation takes longer time to train in comparison to Pytorch's implementation for the same number of epochs. Some of the issues in custom implementation are the usage of for-loops which makes the custom GRU much slower. This can be handled by efficiently computing the gate outputs via. vectorization. Different learning rates could also be used to train different GRU modules which might lead to faster training and better performance.

4 Conclusion

In conclusion, Recurrent Neural Network using GRU is a powerful way to capture sequential text features which are used in sequential data modeling and text classification. It enables the model to learn both long and short-term dependencies of a word context in a text. Appropriate gating mechanisms are also required for training RNNs otherwise the model may face vanishing gradient problems. Hence, well-designed network architectures containing all these layer elements along with appropriate hyper-parameters are essential in making the training much more efficient and stable and getting higher accuracy in text classification.

5 Source Code

```
# -*- coding: utf-8 -*-
"""hw8_SouradipPal.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1NPBvNAl8-unsn4zjLwXH_yOFAAOOTOsR
"""
```

```
from google.colab import drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/

!wget -O DLStudio-2.2.5.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.2.5.tar.gz?download

!tar -xvf DLStudio-2.2.5.tar.gz

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/

!pip install pymsgbox

!python setup.py install

# %load_ext autoreload
# %autoreload 2

import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt

from tqdm import tqdm
import gzip
import pickle

seed = 0
random.seed(seed)
np.random.seed(seed)

from DLStudio import *

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/Examples/

!wget -O /content/text_datasets_for_DLStudio.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/text_datasets_for_DLStudio.tar.gz

!tar -xvf /content/text_datasets_for_DLStudio.tar.gz -C \
    /content/drive/MyDrive/Purdue/ECE60146/DLStudio-2.2.5/Examples/

!tar -xvf data/sentiment_dataset_train_400.tar.gz

!tar -xvf ./data/sentiment_dataset_test_400.tar.gz

!python text_classification_with_GRU_word2vec.py

!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW8/data/
!mkdir /content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models

import gensim.downloader as gen_api
import gensim.downloader as genapi
from gensim.models import KeyedVectors

# Custom SentimentAnalysis Dataset
class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, root, dataset_file, mode = 'train', path_to_saved_embeddings=None):
        super(SentimentAnalysisDataset, self).__init__()
        self.path_to_saved_embeddings = path_to_saved_embeddings
        self.mode = mode
        root_dir = root
        f = gzip.open(root_dir + dataset_file, 'rb')
        dataset = f.read()
        if path_to_saved_embeddings is not None:
            if os.path.exists(path_to_saved_embeddings + 'vectors.kv'):
                self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv')
            else:
                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 mode == '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 mode == '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_tensor = torch.zeros(2)
        if sentiment == 1:
            sentiment_tensor[1] = 1
        elif sentiment == 0:
            sentiment_tensor[0] = 1
        sentiment_tensor = sentiment_tensor.type(torch.long)
        return sentiment_tensor

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

    def __getitem__(self, idx):
        sample = self.indexed_dataset_train[idx] if self.mode == 'train' else self.indexed_dataset_test[idx]
        review = sample[0]
        review_category = sample[1]
        review_sentiment = sample[2]
        review_sentiment = self.sentiment_to_tensor(review_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

# Create custom training dataset
```

```
train_dataset = SentimentAnalysisDataset('./data/','sentiment_dataset_train_400.tar.gz',
                                         path_to_saved_embeddings = './data/word2vec/')

len(train_dataset)

# Create custom validation dataset
val_dataset = SentimentAnalysisDataset('./data/', 'sentiment_dataset_test_400.tar.gz',
                                       mode = 'test', path_to_saved_embeddings = './data/word2vec/')

len(val_dataset)

# Create custom training/validation dataloader
train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=1, shuffle=True, num_workers=1)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=1, shuffle=False, num_workers=1)

# Check dataloader
train_loader_iter = iter(train_data_loader)
sample = next(train_loader_iter)

print('sample has length: ', len(sample))
#print('target has length: ', len(label))

sample

import seaborn as sns

# Plotting confusion matrix
def plot_conf_mat(conf_mat, classes, model_name):
    labels = []
    num_classes = len(classes)
    for row in range(num_classes):
        rows = []
        total_labels =  np.sum(conf_mat[row])
        for col in range(num_classes):
            count = conf_mat[row][col]
            percent = "%.2f%%" % (count / total_labels * 100)
            label = str(count) + '\n' + str(percent)
            rows.append(label)
        labels.append(rows)
    labels = np.asarray(labels)

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

# Commented out IPython magic to ensure Python compatibility.
import torch.nn as nn
import torch.optim as optim
import copy
import time

# Routine to train GRU with embeddings
def train_gru_with_embeddings(device, net, dataloader, model_name, epochs, display_interval):
    #net = copy.deepcopy(net)
    net = net.to(device)

    criterion = nn.NLLLoss()
    accum_times = []
    optimizer = optim.Adam(net.parameters(), lr=1e-3, betas = (0.5, 0.999))
    training_loss_tally = []
    start_time = time.perf_counter()
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(dataloader):
            review_tensor,category,sentiment = data['review'], data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            optimizer.zero_grad()
            hidden = net.init_hidden().to(device)
            output, hidden = net(torch.unsqueeze(review_tensor[0], 1), hidden)
            loss = criterion(output, torch.argmax(sentiment, 1))
```

```
            running_loss += loss.item()
            loss.backward()
            optimizer.step()

            if (i+1) % display_interval == 0:
                avg_loss = running_loss / float(display_interval)
                training_loss_tally.append(avg_loss)
                current_time = time.perf_counter()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time:%4d secs] loss: %.5f"
#                       % (epoch+1,i+1, time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                running_loss = 0.0

    # Save model weights
    checkpoint_path = os.path.join('/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models',
                                   f'{model_name}.pt')
    torch.save(net.state_dict(), checkpoint_path)

    print("Total Training Time: {}".format(str(sum(accum_times))))
    print("\nFinished Training\n\n")
    plt.figure(figsize=(10,5))
    plt.title(f"Training Loss vs. Iterations - {model_name}")
    plt.plot(training_loss_tally)
    plt.xlabel("Iterations")
    plt.ylabel("Training loss")
    plt.legend()
    plt.savefig(f"/content/drive/MyDrive/Purdue/ECE60146/HW8/data/training_loss_{model_name}.png")
    plt.show()

    return training_loss_tally

# Routine to validate GRU with embeddings
def validate_gru_with_embeddings(device, net, model_path, dataloader, model_name, display_interval):
    net.load_state_dict(torch.load(model_path))
    net = net.to(device)
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    confusion_matrix = torch.zeros(2,2)
    with torch.no_grad():
        for i, data in enumerate(dataloader):
            review_tensor,category,sentiment = data['review'], data['category'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)

            hidden = net.init_hidden().to(device)
            output, hidden = net(torch.unsqueeze(review_tensor[0], 1), hidden)
            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if (i+1) % display_interval == 0:
                print("   [i=%d]    predicted_label=%d       gt_label=%d" % (i+1, predicted_idx,gt_idx))
            if predicted_idx == gt_idx:
                classification_accuracy += 1
            if gt_idx == 0:
                negative_total += 1
            elif gt_idx == 1:
                positive_total += 1
            confusion_matrix[gt_idx,predicted_idx] += 1
    print("\nOverall classification accuracy: %0.2f%%" %  (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)
```

```
    plot_conf_mat(confusion_matrix.numpy(), ['negative_reviews', 'positive_reviews'], model_name)

# Plotting training loss
def plot_loss(mean_losses, display_interval, model_name):
    plt.figure()
    plt.plot(np.arange(len(mean_losses))*display_interval, mean_losses, label="Training Loss");
    plt.title(f'Training Loss for {model_name}')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

# torch.nn GRU with Embeddings
class GRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(GRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_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(  2,          1,         self.hidden_size    ).zero_()
        return hidden

# Initialize device
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
device

# Initialize torch.nn GRU with Embeddings
model = GRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2, num_layers=2)

# model.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_nnGRU.pt'))

epochs = 5
display_interval = 500

number_of_learnable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

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

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

# Train nnGRU
net1_losses = train_gru_with_embeddings(device, model, dataloader = train_data_loader,
                        model_name = 'HW8_nnGRU', epochs=epochs, display_interval = display_interval)

# Validate nnGRU
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_nnGRU.pt'
validate_gru_with_embeddings(device, model, dataloader = val_data_loader,
             display_interval = display_interval, model_path = save_path, model_name = 'nnGRU')

# GRU Cell
from torch.autograd import Variable

class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size, bias=False):
        super(GRUCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias

        self.input2hidden = nn.Linear(input_size, 3 * hidden_size, bias=bias) # [W_z, W_r, W_h]
```

```
        self.hidden2hidden = nn.Linear(hidden_size, 3 * hidden_size, bias=bias) # [U_z, U_r, U_h]

        self.reset_parameters()

    def reset_parameters(self):
        std = 1.0 / np.sqrt(self.hidden_size)
        for w in self.parameters():
            w.data.uniform_(-std, std)

    def forward(self, input, hx=None):
        if hx is None:
            hx = Variable(input.new_zeros(input.size(0), self.hidden_size))

        x_t = self.input2hidden(input)
        h_t = self.hidden2hidden(hx)

        x_reset, x_update, x_new = x_t.chunk(3, 1)
        h_reset, h_update, h_new = h_t.chunk(3, 1)

        reset_gate = torch.sigmoid(x_reset + h_reset) #reset gate
        update_gate = torch.sigmoid(x_update + h_update) #update gate
        new_gate = torch.tanh(x_new + (reset_gate * h_new)) #candidate hidden state

        hy = update_gate * hx + (1 - update_gate) * new_gate

        return hy

# Custom GRU
class CustomGRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, bias = False):
        super(CustomGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bias = bias

        self.cell_list = nn.ModuleList()
        self.cell_list.append(GRUCell(self.input_size, self.hidden_size, self.bias))
        for l in range(1, self.num_layers):
            self.cell_list.append(GRUCell(self.hidden_size, self.hidden_size, self.bias))

    def forward(self, input, hx=None):
        if hx is None:
            if torch.cuda.is_available():
                h0 = Variable(torch.zeros(self.num_layers, input.size(1), self.hidden_size).cuda())
            else:
                h0 = Variable(torch.zeros(self.num_layers, input.size(1), self.hidden_size))
        else:
             h0 = hx

        outs = []
        hidden = list()

        for layer in range(self.num_layers):
            hidden.append(h0[layer, :, :])

        for t in range(input.size(0)):
            for layer in range(self.num_layers):
                if layer == 0:
                    hidden_out = self.cell_list[layer](input[t, :, :], hidden[layer])
                else:
                    hidden_out = self.cell_list[layer](hidden[layer - 1], hidden[layer])
                hidden[layer] = hidden_out

            outs.append(hidden_out)

        return outs, hidden

# Custom GRUnetWithEmbeddings
class CustomGRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(CustomGRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = CustomGRU(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(  2,          1,         self.hidden_size    ).zero_()
        return hidden

# Initialize CustomGRU with Embeddings
model2 = CustomGRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2, num_layers=2)

# model.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_customGRU.pt'))

epochs = 5
display_interval = 500

number_of_learnable_params = sum(p.numel() for p in model2.parameters() if p.requires_grad)

num_layers = len(list(model2.parameters()))

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

# Training custom GRU
net2_losses = train_gru_with_embeddings(device, model2, dataloader = train_data_loader,
                      model_name = 'HW8_customGRU', epochs=epochs, display_interval = display_interval)

# Validate custom GRU
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_customGRU.pt'
validate_gru_with_embeddings(device, model2, dataloader = val_data_loader,
             display_interval = display_interval, model_path = save_path, model_name = 'customGRU')

# Bidirectional torch.nn GRU with Embeddings
class BiGRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(BiGRUnetWithEmbeddings, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers, bidirectional=True)
        self.fc = nn.Linear(2*hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

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

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

# Initialize Bidirectional torch.nn GRU with Embeddings
model3 = BiGRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2, num_layers=2)

model3.load_state_dict(torch.load('/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_nnBiGRU.pt'))

number_of_learnable_params = sum(p.numel() for p in model3.parameters() if p.requires_grad)

num_layers = len(list(model3.parameters()))

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

```
epochs = 5
display_interval = 500

# Training Bidirectional nnGRU
net3_losses = train_gru_with_embeddings(device, model3, dataloader = train_data_loader,
                        model_name = 'HW8_nnBiGRU', epochs=epochs, display_interval = display_interval)

# Validate Bidirectional GRU
save_path = '/content/drive/MyDrive/Purdue/ECE60146/HW8/saved_models/HW8_nnBiGRU.pt'
validate_gru_with_embeddings(device, model3, dataloader = val_data_loader,
             display_interval = display_interval, model_path = save_path, model_name = 'nnBiGRU')
```