BME 646/ ECE695DL: Homework 8

Bianjiang Yang

26 April 2022

1 Introduction

The main goal for homework6:
1. To gain insights into the workings of Recurrent Neural Networks. These are neural networks with feedback. We need such networks for language modeling, sequence-to-sequence learning (as in automatic translation systems), time-series data prediction, etc.
2. To understand the performance quality variations with various levels of gating.
3. To use RNNs for the modeling of variable-length product reviews provided by Amazon for automatic classification of such reviews.

2 Methodology

Packages: torch, torch.nn, torch.nn.functional, matplotlib.pyplot, copy, os, sys, glob, time, numpy, gzip, re, pickle
Language: Python3
Tools: Anaconda3 virtual environment
System: Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-43-generic x86_64)

Instructions for running the code: The code I submitted has been modified to be runable, so you can simply run it by excuting the following script:
(1)train_GRU.py: CUDA_VISIBLE_DEVICES=1 python train_GRU.py
(2)train_pmGRU.py: CUDA_VISIBLE_DEVICES=1 python train_pmGRU.py
Note: "train_GRU.py" and "train_pmGRU.py" will also implement testing after training. "validation.py" is not runnable.
(3) You can also test the model later by running "test_GRU.py" and "test_pmGRU.py".
Other information: The code is based on DLStudio's code.

3 Implementation and Results

network.py

```
import torch.nn as nn
import torch.nn.functional as F
import torch

class GRUNet(nn.Module):
    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 word2ve
               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(GRUNet, 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



class pmGRU(nn.Module):
    """
    This GRU implementation is based primarily on a "Minimal Gated" version of a GRU as described in
    "Simplified Minimal Gated Unit Variations for Recurrent Neural Networks" by Joel Heck and Fathi
    Salem. The Wikipedia page on "Gated_recurrent_unit" has a summary presentation of the equations
    proposed by Heck and Salem.
    """

    def __init__(self, input_size, hidden_size, output_size, batch_size):
```

```
        super(pmGRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.batch_size = batch_size
        ## for forget gate:
        self.project1 = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size)
        ## for interim out:
        self.project2 = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size)
        ## for final out
        self.project3 = nn.Sequential(nn.Linear(self.hidden_size, self.output_size), nn.Tanh())

        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        combined1 = torch.cat((x, h), -1)
        forget_gate = self.project1(combined1)
        interim = forget_gate * h
        combined2 = torch.cat((x, interim), -1)
        output_interim = self.project2(combined2)
        hid = (1 - forget_gate) * h + forget_gate * output_interim
        output =self.fc(self.relu(hid[:,-1]))
        output = self.logsoftmax(output)
        return output, hid

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

dataloader.py

```
import torch
import glob
import gzip
import sys,os,os.path
import torch
# import torch.nn as nn
import torch.nn.functional as F
# import torchvision
# import torchvision.transforms as tvt
```

```
# import torch.optim as optim
import numpy as np
# import numbers
import re
# import math
import random
# import copy
# import matplotlib.pyplot as plt
import pickle
# import pymsgbox
import time
# import logging

class SentimentAnalysisDataset(torch.utils.data.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, dataroot, 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 = dataroot
        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
                      """\na couple of minutes. The embeddings occupy around 3.6GB of your disk space
                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(d
            else:
```

```
            self.positive_reviews_train, self.negative_reviews_train, self.vocab = pickle.loads(d
        self.categories = sorted(list(self.positive_reviews_train.keys()))
        self.category_sizes_train_pos = {category : len(self.positive_reviews_train[category]) fo
        self.category_sizes_train_neg = {category : len(self.negative_reviews_train[category]) fo
        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(dat
        else:
            self.positive_reviews_test, self.negative_reviews_test, self.vocab = pickle.loads(dat
        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
        self.category_sizes_test_neg = {category : len(self.negative_reviews_test[category]) for
        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_d
        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_GRU.py

```
import torch
import torch.nn as nn
import network
import matplotlib.pyplot as plt
import dataloader
import copy
import time
import validation


def run_code_for_training_for_text_classification_with_GRU_word2vec(epochs, device, learning_rate, mo
```

```
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 GREnet now produces the LogSoftmax output:
criterion = nn.NLLLoss()
accum_times = []
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
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), hid
        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(sum(accum_times))))
print("\nFinished Training\n\n")
if display_train_loss:
    plt.figure(figsize=(10, 5))
    plt.title("GRU Training Loss vs. Iterations")
    plt.plot(training_loss_tally)
    plt.xlabel("iterations")
```

```
        plt.ylabel("training loss")
        plt.legend()
        plt.savefig("training_loss_GRU1e-4.png")
        plt.show()

if __name__ == '__main__':
    device  = torch.device('cuda:0')
    dataroot = "./DLStudio-2.2.2/Examples/data/"
    dataset_archive_train = "sentiment_dataset_train_200.tar.gz"
    dataset_archive_test = "sentiment_dataset_test_200.tar.gz"
    path_to_saved_embeddings = "/home/yangbj/695/hw8/word2vec/"
    batch_size = 1
    dataserver_train = dataloader.SentimentAnalysisDataset(
        train_or_test='train',
        dataroot=dataroot,
        dataset_file=dataset_archive_train,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    dataserver_test = dataloader.SentimentAnalysisDataset(
        train_or_test='test',
        dataroot=dataroot,
        dataset_file=dataset_archive_test,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                                                batch_size=batch_size, shuffle=True,
                                                num_workers=2)
    test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                               batch_size=batch_size, shuffle=False,
                                               num_workers=2)

    model = network.GRUNet(input_size=300, hidden_size=100, output_size=2, num_layers=2)

    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)

    ## TRAINING:
    print("\nStarting training\n")
    run_code_for_training_for_text_classification_with_GRU_word2vec(epochs=5, device=device, learning
                                                                     momentum = 0.9, train_dataloader=
                                                                     path_saved_model = "./saved_GRU1e

    ## TESTING:
    print("\nStarting testing\n")
```

```
    validation.run_code_for_testing_text_classification_with_GRU_word2vec(path_saved_model = "./saved
```

train_pmGRU.py

```
import torch
import torch.nn as nn
import network
import matplotlib.pyplot as plt
import dataloader
import copy
import time
import validation


def run_code_for_training_for_text_classification_with_GRU_word2vec(epochs, device, learning_rate, mo
    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 GREnet now produces the LogSoftmax output:
    criterion = nn.NLLLoss()
    accum_times = []
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
    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), hid
            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(sum(accum_times))))
    print("\nFinished Training\n\n")
    if display_train_loss:
        plt.figure(figsize=(10, 5))
        plt.title("pmGRU Training Loss vs. Iterations")
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")
        plt.legend()
        plt.savefig("training_loss_pmGRU1e-4.png")
        plt.show()

if __name__ == '__main__':
    device  = torch.device('cuda:0')
    dataroot = "./DLStudio-2.2.2/Examples/data/"
    dataset_archive_train = "sentiment_dataset_train_200.tar.gz"
    dataset_archive_test = "sentiment_dataset_test_200.tar.gz"
    path_to_saved_embeddings = "/home/yangbj/695/hw8/word2vec/"
    batch_size = 1
    dataserver_train = dataloader.SentimentAnalysisDataset(
        train_or_test='train',
        dataroot=dataroot,
        dataset_file=dataset_archive_train,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    dataserver_test = dataloader.SentimentAnalysisDataset(
        train_or_test='test',
        dataroot=dataroot,
        dataset_file=dataset_archive_test,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                                                   batch_size=batch_size, shuffle=True,
```

```
                                                  num_workers=2)
    test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                                 batch_size=batch_size, shuffle=False,
                                                 num_workers=2)

    model = network.pmGRU(input_size=300, hidden_size=100, output_size=2, batch_size=1)

    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)

    ## TRAINING:
    print("\nStarting training\n")
    run_code_for_training_for_text_classification_with_GRU_word2vec(epochs=5, device=device, learning
                                                                    momentum = 0.9, train_dataloader=
                                                                    path_saved_model = "./saved_pmGRU

    ## TESTING:
    print("\nStarting testing\n")
    validation.run_code_for_testing_text_classification_with_GRU_word2vec(path_saved_model = "./saved
```

validation.py

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import seaborn

def run_code_for_testing_text_classification_with_GRU_word2vec(path_saved_model, test_dataloader, net
    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), hid
```

```
            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
            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

    # plot confusion matrix
    plt.figure(figsize=(10, 7))
    seaborn.heatmap(confusion_matrix, annot=True, linewidths=.5,
                    xticklabels=['predicted negative', 'predicted positive'], yticklabels=['true nega
    plt.title("Net" + 'Accuracy:' + str(float(classification_accuracy) * 100 / float(i)) + '%')
    plt.savefig("net_confusion_matrix_GRU.png")

    print("\nOverall classification accuracy: %0.2f%%" % (float(classification_accuracy) * 100 / floa
    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)
```

test_GRU.py

```
import torch
import torch.nn as nn
import network
import matplotlib.pyplot as plt
```

```
import dataloader
import copy
import time
import validation

if __name__ == '__main__':
    device  = torch.device('cuda:0')
    dataroot = "./DLStudio-2.2.2/Examples/data/"
    dataset_archive_test = "sentiment_dataset_test_200.tar.gz"
    path_to_saved_embeddings = "/home/yangbj/695/hw8/word2vec/"
    batch_size = 1
    dataserver_test = dataloader.SentimentAnalysisDataset(
        train_or_test='test',
        dataroot=dataroot,
        dataset_file=dataset_archive_test,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                                  batch_size=batch_size, shuffle=False,
                                                  num_workers=2)

    model = network.GRUNet(input_size=300, hidden_size=100, output_size=2, num_layers=2)

    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)

    ## TESTING:
    print("\nStarting testing\n")
    validation.run_code_for_testing_text_classification_with_GRU_word2vec(path_saved_model = "./saved
```

test_pmGRU.py

```
import torch
import torch.nn as nn
import network
import matplotlib.pyplot as plt
import dataloader
import copy
import time
import validation
```

```
if __name__ == '__main__':
    device  = torch.device('cuda:0')
    dataroot = "./DLStudio-2.2.2/Examples/data/"
    dataset_archive_test = "sentiment_dataset_test_200.tar.gz"
    path_to_saved_embeddings = "/home/yangbj/695/hw8/word2vec/"
    batch_size = 1
    dataserver_test = dataloader.SentimentAnalysisDataset(
        train_or_test='test',
        dataroot=dataroot,
        dataset_file=dataset_archive_test,
        path_to_saved_embeddings=path_to_saved_embeddings,
    )
    test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                                  batch_size=batch_size, shuffle=False,
                                                  num_workers=2)

    model = network.pmGRU(input_size=300, hidden_size=100, output_size=2, batch_size=1)

    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)

    ## TESTING:
    print("\nStarting testing\n")
    validation.run_code_for_testing_text_classification_with_GRU_word2vec(path_saved_model = "./saved
```

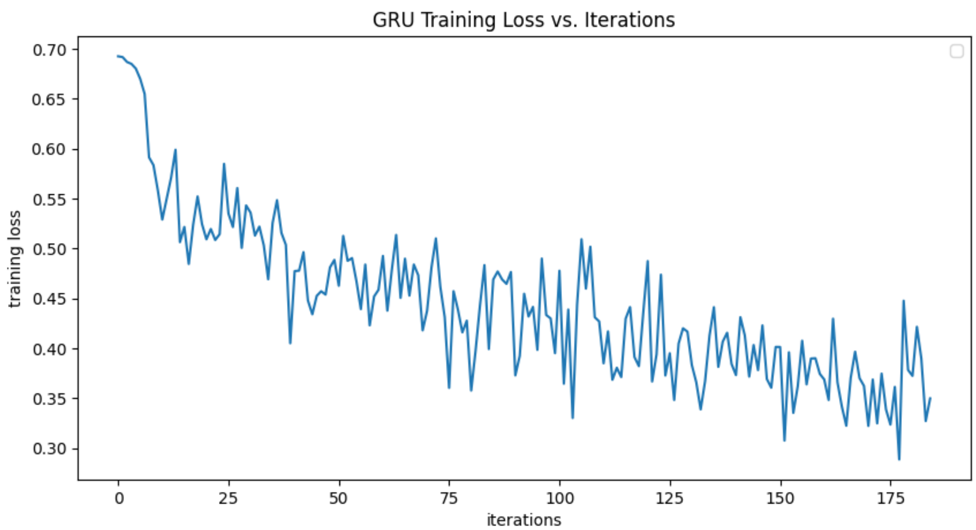

Figure 1: GRU loss

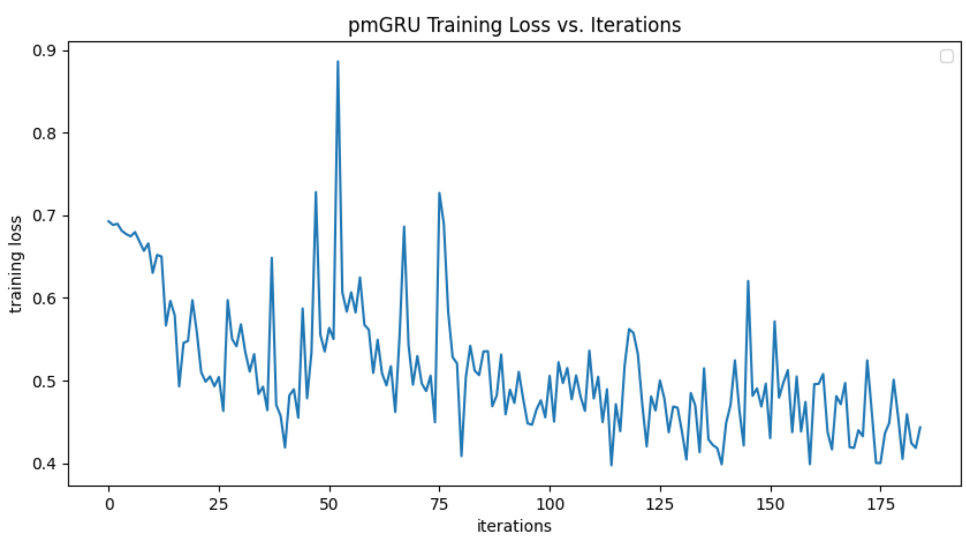

Figure 2: pmGRU loss

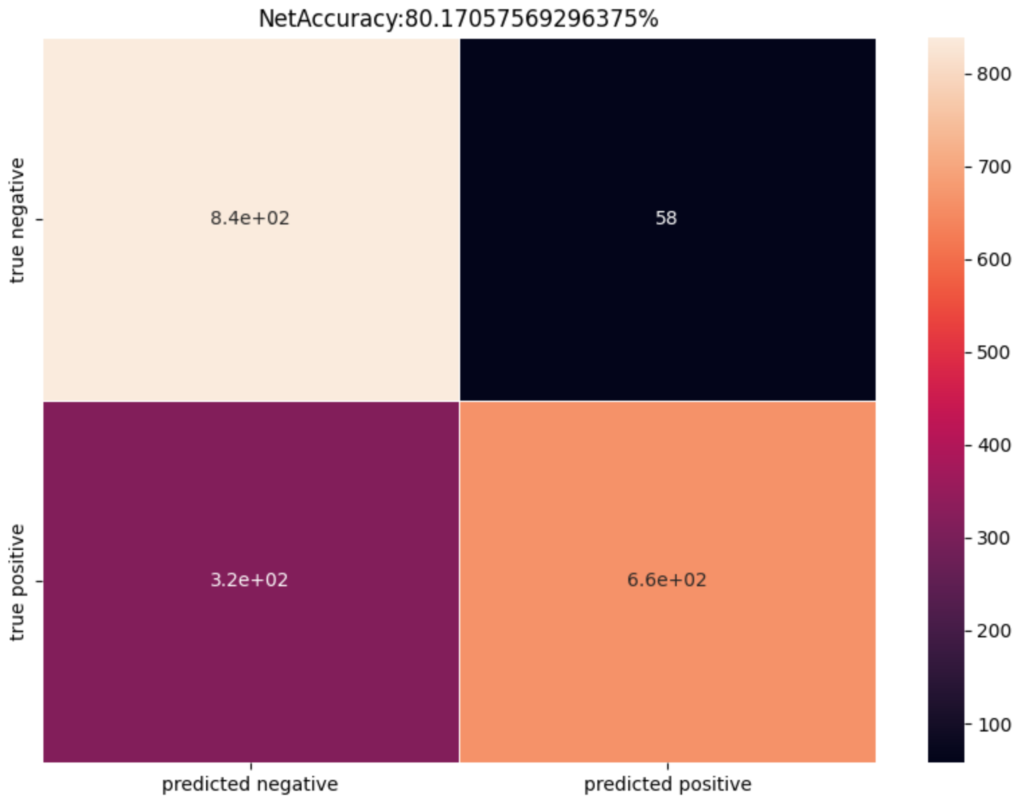

Figure 3: GRU: confusion matrix

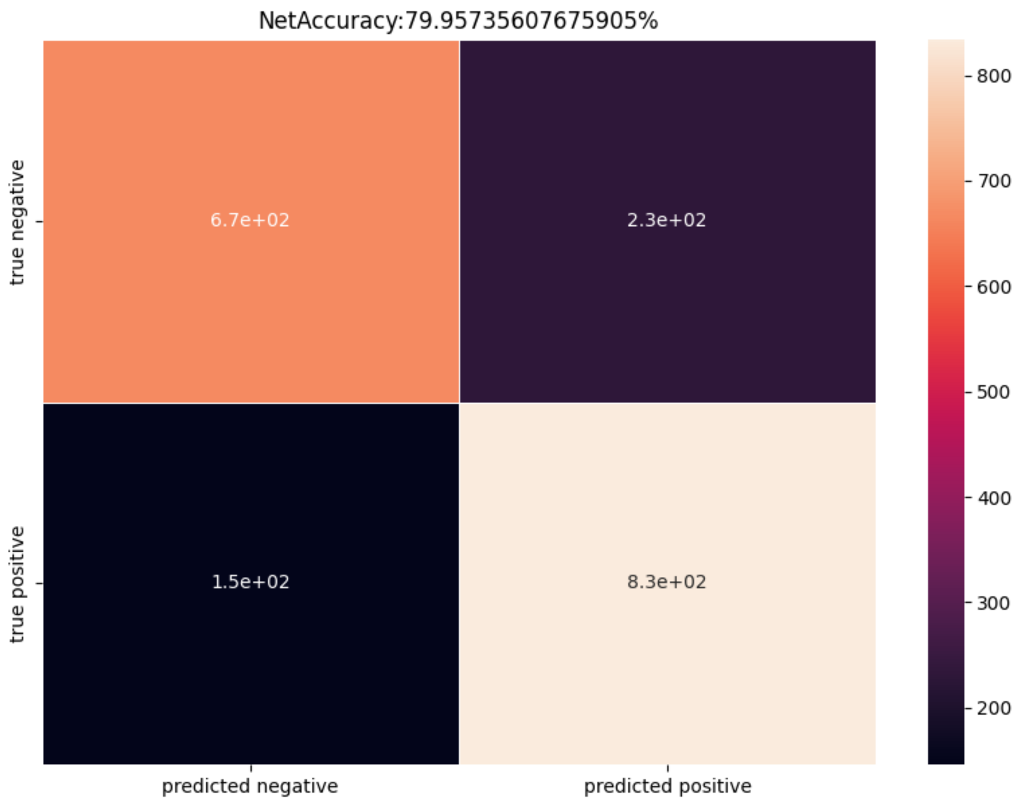

Figure 4: pmGRU: confusion matrix

Train Loss
The results in Fig. **??** and Fig. **??** above show that:
(1) The loss is decreasing overall
(2) The decreasing tendency of GRU is more obvious than that of pmGRU.
(3) The final loss value of pmGRU is larger than that of GRU.

Confusion Matrix
The results in Fig. **??** and Fig. **??** above shows that:
(1) The GRU has an accuracy of 80.17%, which indicates most of the prediction is correct;
(2) The pmGRU has an accuracy of 79.95%, which is slightly lower than that of GRU, but it also indicates most of the prediction is correct;

GRU Equations

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \tag{1}$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \tag{2}$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{3}$$
$$\tilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t]_j) \tag{4}$$

pmGRU Equations

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) \tag{5}$$
$$\tilde{h}_t = tanh(W_h \cdot x_t + U_h \cdot (f_t * h_{t-1}) \tag{6}$$
$$h_t = (1 - f_t) * h_{t-1} + f_t * \tilde{h}_t \tag{7}$$

From the equations, it is easy to find that pmGRU is a simplied version of GRU. The experiment results also showed that pmGRU has higher loss and lower accuracy with the same training setting.

Implementation Details. Batch size=1; I choose Adam optimizer with learning rate 10^{-4} for more stable and faster training because SGD optimizes too slowly for such learning rate. I trained both of the model for 5 epochs.

Other results. Model complexity of pmGRU:
The number of layers in the model: 8;
The number of learnable parameters in the model: 80604;

Model complexity of GRU:
The number of layers in the model: 10;
The number of learnable parameters in the model: 181402;

4 Lessons Learned

The hurdles faced and the techniques employed to overcome them:

(1) For network,
We need to add an output layer after the pmGRU, which is the same as the GRU's.

(2) For training,
Adam is much better than SGD optimizer. Larger learning rate is preferred (for example 1e-4 is better than 1e-5).

(3) For validation,
We need to eval the model to keep the model parameters static.
seaborn is a better tool than sklearn to plot the heatmap.
Confusion Matrix should be computed on CPU because of sklearn

5 Suggested Enhancements

To the best of my understanding, this task emphasize more on the network design. However, most of the time was consumed in the ploting and some unimportant part debugging. So I suggest you could give a colab file and let the students fill necessary blanks.