HW 8: Gating mechanisms in RNN

David E. Farache, Email ID: dfarache@purdue.edu

April 18, 2023

1 Introduction

The focus of this assignment was the utilization of Recurrent Neural Networks (RNN) with gating mechanisms using a Gated Recurrent Unit (GRU) that resolves the vanishing gradient problem. This is paired with word embedding using wrod2vec, which is utilized to distinguish between positive and negative reviews from a given dataset.

2 Explanation of Work

2.1 Word Embedding

Word Embedding is a method to attribute a numeric evaluation to words that pertains to a similar significance to words within a repository that possess similar meaning. In this case, we use the Word2Vec method, which grants word embedding by feeding a one-hot encoding into a neural network. The one-hot encoding is obtained by scanning of the text with a window size of 2W+1 with the central word being the focus word and the surroundings being categorized as context words. The focus word is fed as a one-hot encoding of vocabulary size V into the neural network which the first linear layer turns into a projection by multiplying by a matrix W of learnable parameters. That projection is then fed into another neural network with the SoftMax activation function in order to extract the conditional probability of each node of the vocab being the context words given as the focus word we isolated earlier.

In this case, word2vec embedding is from google news and each item is saved in the dictionary with the category assigned and its ground truth for the dataloader.

2.2 RNN

RNN is a certain neural network method that enables the passing of prior inference to the following layer, creating a sense of "memory" that enables, meaning sequential and time series data passes. This is important in natural language processing as the context within a sentence is critical to comprehend the actual meaning of words and if passing individual vocab, there needs to be a method to have a memory attached to that. The weakness of the RNN is that based on the number of iterations, one can quickly run into the vanishing gradient problem as short-term dependencies dominate instead of long-term.

2.3 GRU

A GRU cell resolves the issue of the vanishing gradient problem as it introduces long-term memory or information into the network. This idea is that a cell retains the prior information which can be updated if deemed important and added to the network if the memory currently saved is considered relevant. In the case of GRU, the values presenting these are x, the inputs, and h, the hidden state of x. The value passes through if the gate value, z, is 0 and if 1 would be temporarily saved, the binary valuation is given by a sigmoid activation function:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{1}$$

A reset gate is added on to decide whether to forget and creates a candidate hidden state, that reserves a certain amount of information from previous hidden states as described below:

$$z_r = \sigma(W_r x_t + U_r h_{t-1}) \tag{2}$$

$$\tilde{h} = tanh(W_h x_t + U_h(r_t \odot ht - 1) \tag{3}$$

The hidden state is updated using the equation below:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{4}$$

In the case of coding the network, afterword embedding is fed into the network as inputs the output features are set to three times that of the hidden features which are then chunked into three parameters that are set equal to the hidden size. This is fed into the linear layer that consolidates results to be passed into the activation function and return probability for the sentiment of the class.

2.4 Network, Training

All networks were run for 5 epochs, batch size of 1, input size of 300, hidden size of 100, output size of 2 (as there were looking for positive or negative reviews), learning rate 1e-4, and trained used the Adam optimizer.

3 Setup and Wrod2Vec

3.1 Setup Code

```
from torch.functional import Tensor

#Import
import os
import sys
import random
import json
import numpy as np
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```

from torch.utils.data import DataLoader , Dataset
import torchvision.transforms as tvt
from PIL import Image
import requests
from requests.exceptions import ConnectionError , ReadTimeout , TooManyRedirects , MissingSchema , InvalidURL
from pycocotools.coco import COCO
import copy
import pickle
import gzip
import matplotlib.pyplot as plt
import logging
import glob
import torchvision.transforms.functional as tvtF
import scipy
import gensim.downloader as gen_api
from gensim.models import KeyedVectors
import time

device = 'cuda'
device = torch.device(device)

root_dir = "/scratch/gilbreth/dfarache/ece60146/David/HW8/"
path_to_saved_embeddings = "/scratch/gilbreth/dfarache/ece60146/David/HW8/word2vec/"

train_dataset_file = "sentiment_dataset_train_400.tar.gz"
test_dataset_file = "sentiment_dataset_test_400.tar.gz"

batch_size = 1
num_layers = 1

class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, root_dir, train_or_test, dataset_file, path_to_saved_embeddings):
        super(SentimentAnalysisDataset, self).__init__()

        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

        f = gzip.open(root_dir + dataset_file, 'rb')
        dataset = f.read()

        self.indexed_dataset_train = []
        self.indexed_dataset_test = []

        self.load_in_dataset(dataset)

        if train_or_test == 'train':
            self.indexed_dataset_train = self.indexed_dataset
        elif train_or_test == 'test':
            self.indexed_dataset_test = self.indexed_dataset

    def load_word_vector(self):
        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')

    def load_in_dataset(self, dataset):
        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 = []

        for category in self.positive_reviews_test:
            for review in self.positive_reviews_test[category]:
                self.indexed_dataset.append([review, category, 1])

        for category in self.negative_reviews_test:
            for review in self.negative_reviews_test[category]:
                self.indexed_dataset.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)

        # Conver to one-hot encoding
        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
```

Listing 1: *Setup Code*

3.2 Task 1: GRU Network From Scratch

For the GRU logic, the input and hidden state are concatenated if hx is fed into the model. The data is then passed into a linear layer and evaluated via a sigmoid activation function for the reset and update gate. New data is then given by concatenation of the input and the Hadmard product of the reset and hidden gate via a linear layer and the tanh activation function. The following hidden state is then found by the sum of two Hadard products, one of the past hidden state and (1-z), and the second the update gate and new data. This resolves the vanishing gradient problem in a similar matter of a skip-block as the update gate, passes prior information and the reset gate then grants what is forgotten within the hidden state.

```
# GRU Net Homebrew
# Based on https://github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py

```

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

        self.x2h = nn.Linear(input_size, 3 * hidden_size, bias=bias)
        self.h2h = nn.Linear(hidden_size, 3 * hidden_size, bias=bias)

        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, inputs, hx=None):
        if(hx is None):
            hx = torch.zeros((batch_size, self.hidden_size), device=device, dtype=X.dtype, requires_grad=True)

        x_t = self.x2h(inputs)
        h_t = self.h2h(hx)

        x_reset, x_upd, x_new = x_t.chunk(3, 1)
        h_reset, h_upd, h_new = h_t.chunk(3, 1)

        reset_gate = torch.sigmoid(x_reset + h_reset)
        update_gate = torch.sigmoid(x_upd + h_upd)
        new_gate = torch.tanh(x_new + (reset_gate * h_new))

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

        return hy
```

Listing 2: *GRU Cell*

3.3

```python
# Based on github.com/georgeyiasemis/Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py
class GRUNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, bias=True):
        super(GRUNetwork, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bias = bias
        self.output_size = output_size

        self.rnn_cell_list = nn.ModuleList()
        self.rnn_cell_list.append(GRUCell(self.input_size,
                                          self.hidden_size,
```

```
                                                  self.bias))

        self.logSoftMax = nn.LogSoftmax()

        for layer in range(1, self.num_layers):
            self.rnn_cell_list.append(GRUCell(self.input_size,
                                              self.hidden_size,
                                              self.bias))
        self.fc = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, inputs, hx=None):
        if(hx is None):
            hx = torch.zeros((self.num_layers, batch_size, self.hidden_size),
    device=device, dtype=inputs.dtype, requires_grad=True)

        outs = []
        hidden = []
        for layer in range(self.num_layers):
            hidden.append(hx[layer, :, :])

        for t in range(inputs.shape[1]):
            for layer in range(self.num_layers):

                if(not layer):
                    hidden_layer = self.rnn_cell_list[layer](inputs[:, t, :],
    hidden[layer])
                else:
                    hidden_layer = self.rnn_cell_list[layer](hidden[layer - 1],
    hidden[layer])

                hidden[layer] = hidden_layer
            outs.append(hidden_layer)

        outs = outs[-1]
        outs = self.fc(outs)
        outs = self.logSoftMax(outs)

        return outs
```

Listing 3: *GRU Network*

4

4.1 Task 2: GRU Network Bidirectional Testing

```
# PyTorch GRU Net
# Based on DLStudio network
class GRUnetWithEmbeddings(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, bidirectional_flag,
     num_layers=1):

        super(GRUnetWithEmbeddings, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
```

```
        self.output_size = output_size
        self.num_layers = num_layers

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

        if bidirectional_flag: self.flag_value = 2
        else: self.flag_value = 1

        self.fc = nn.Linear(hidden_size * self.flag_value, 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
        hidden = weight.new_zeros((self.num_layers * self.flag_value, batch_size, self.hidden_size))
        return hidden
```

Listing 4: *Bidirectional GRU Network*

4.2 Training, Testing, and Plotting

```
# Training
# Based on DLStudio network
def training_classification_with_GRU_word2vec(net, train_dataloader, lr, betas, epochs, save_model, task, log=800):
    net = net.to(device)

    ##  Note that the GRUnet now produces the LogSoftmax output:
    criterion = nn.NLLLoss()
    accum_times = []
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas) # Adam Optimizer

    training_loss_tally = []
    start_time = time.time()

    for epoch in range(epochs):
        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)
            category = category.to(device)

            optimizer.zero_grad()
```

```
            if task=="1":
                hidden = net.to(device)
                output = net(review_tensor)
            else:
                hidden = net.init_hidden().to(device)
                output, hidden = net(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.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))

                running_loss = 0.0

                torch.save(net.state_dict(), os.path.join(root_dir + "model/", save_model))

    print("Total Training Time: {}".format(str(sum(accum_times))))
    print("\nFinished Training\n\n")

    return net, training_loss_tally
```

Listing 5: *Training*

```
# Testing
# Based on DLStudio network
def testing_text_classification_with_GRU_word2vec(test_dataloader, net, save_model, task):

    classification_accuracy = 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']

            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            category = category.to(device)
            if task=="1":
```

```
                hidden = net.to(device)
                output = net(review_tensor)
            else:
                hidden = net.init_hidden().to(device)
                output, hidden = net(review_tensor, hidden)

            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()

            #Update per step
            if i % 100 == 99:
                print("   [i=%d]    predicted_label=%d       gt_label=%d" % (i+1, predicted_idx,gt_idx))

            # Get accuracy
            if predicted_idx == gt_idx:
                classification_accuracy += 1

            # Count negative reviews
            if gt_idx == 0:
                negative_total += 1

            # Count positive reviews
            elif gt_idx == 1:
                positive_total += 1

            confusion_matrix[gt_idx,predicted_idx] += 1

    # Display results
    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))

    out_str = "                       "
    out_str +=  "%18s    %18s" % ('predicted negative', 'predicted positive')
    print(out_str + "\n")

    acc = (float(classification_accuracy) * 100 /float(i))
    for i,label in enumerate(['true negative', 'true positive']):
        out_str = "%12s:  " % label

        for j in range(2):
            out_str +=  "%18s%%" % out_percent[i,j]

        print(out_str)

    return confusion_matrix, acc
```

Listing 6: *Testing*

```python
# Plotting
def plot_losses(loss, epochs, mode="scratch"):
    plt.figure(figsize=(10,5))

    iterations = range(len(loss))
    plt.plot(iterations, loss)

    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.legend()

    filename = "train_loss_" + mode + ".jpg"
    plt.show()
def display_confusion_matrix(conf, accuracy, class_list, task, bidirectional="
    no_bid"):
    plt.figure(figsize=(10,5))
    sns.heatmap(conf, xticklabels=class_list, yticklabels=class_list, annot=True
    )
    plt.xlabel("True Label \n Accuracy: %0.2f%%" % accuracy)
    plt.ylabel("Predicted Label")
```

Listing 7: *Plotting*

5 Results

```python
# Run Code
batch_size = 1

train_dataset = SentimentAnalysisDataset(root_dir, 'train', train_dataset_file,
    path_to_saved_embeddings)
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=
    batch_size, shuffle=True, num_workers=2, drop_last=True)

test_dataset = SentimentAnalysisDataset(root_dir, 'test', test_dataset_file,
    path_to_saved_embeddings)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=
    batch_size, shuffle=True, num_workers=2, drop_last=True)
# Task 1
# Parameters for training
lr = 1e-4 # Learning Rate
betas = (0.9, 0.999) # Betas factor
epochs = 5 # Number of epochs to train
gru_scratch = GRUNetwork(input_size=300, hidden_size=100, output_size=2,
    num_layers=num_layers)
# Train Model
net_gru_scratch, training_loss_gru_scratch =
    training_classification_with_GRU_word2vec(gru_scratch,
                        train_dataloader, lr, betas, epochs, "
    bidirectional_true_model", "1")


# Test
conf_matrix_gru_scratch, classification_accuracy_gru_scratch =
    testing_text_classification_with_GRU_word2vec(test_dataloader,
    net_gru_scratch, "bidirectional_true_model", "1")
```

```
# Task 2
gru_net_false = GRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2, num_layers=num_layers, bidirectional_flag=False)
gru_net_true = GRUnetWithEmbeddings(input_size=300, hidden_size=100, output_size=2, num_layers=num_layers, bidirectional_flag=True)
#Yes Bidirectional
# Train Model
net_yes_bidirectional, training_loss_gru_yes = training_classification_with_GRU_word2vec(gru_net_true,
                        train_dataloader, lr, betas, epochs, "bidirectional_true_model", "2")

# Test
conf_matrix_gru_yes , classification_accuracy_gru_yes = testing_text_classification_with_GRU_word2vec(test_dataloader, net_yes_bidirectional, "bidirectional_true_model", "2")
#No Bidirectional
# Train
net_no_bidirectional, training_loss_gru_no = training_classification_with_GRU_word2vec(gru_net_false,
                        train_dataloader, lr, betas, epochs, "bidirectional_false_model", "2")
# Test
conf_matrix_gru_no, classification_accuracy_gru_no = testing_text_classification_with_GRU_word2vec(test_dataloader, net_no_bidirectional, "bidirectional_false_model", "2")

# Plot Loss
plot_losses(training_loss_gru_scratch, epochs, mode="torch_bid")
plot_losses(training_loss_gru_yes, epochs, mode="torch_bid")
plot_losses(training_loss_gru_no, epochs, mode="torch_bid")

# Plot Confusion Matrix
display_confusion_matrix(conf_matrix_gru_scratch, classification_accuracy_gru_scratch, classes, task=1, bidirectional="bid")
display_confusion_matrix(conf_matrix_gru_yes, classification_accuracy_gru_yes, classes, task=2, bidirectional="bid")
display_confusion_matrix(conf_matrix_gru_no, classification_accuracy_gru_no, classes, task=2, bidirectional="bid")
```

Listing 8: *Main To Run Code*

Figures in 1, 2, and 3, are the loss values vs iteration for each type of network created. The GRU network with bidirectional scan has the lowest loss value after training in comparison to it turned off and the scratch network. The scratch network did match or at least provide a similar result as the Pytorch GRU cell. When comparing the accuracy of the models, the network with bidirectional scanning does slightly better than as seen in Figure 7-9. Interestingly the network with the scratch GRU cell performs better than the other two, although again no significant improvement, which indicates that the difference may be the parameters and architecture of the network and not the GRU cell. Another possibility could be noise and that if the networks were rerun, you would see the accuracy of the networks vary. The network with scratch GRU cells did perform better at predicting negatives but works at positive reviews. Another thing to note is that the GRU cell made from scratch took 2 hours to train while the PyTorch implementation, took a couple of minutes.

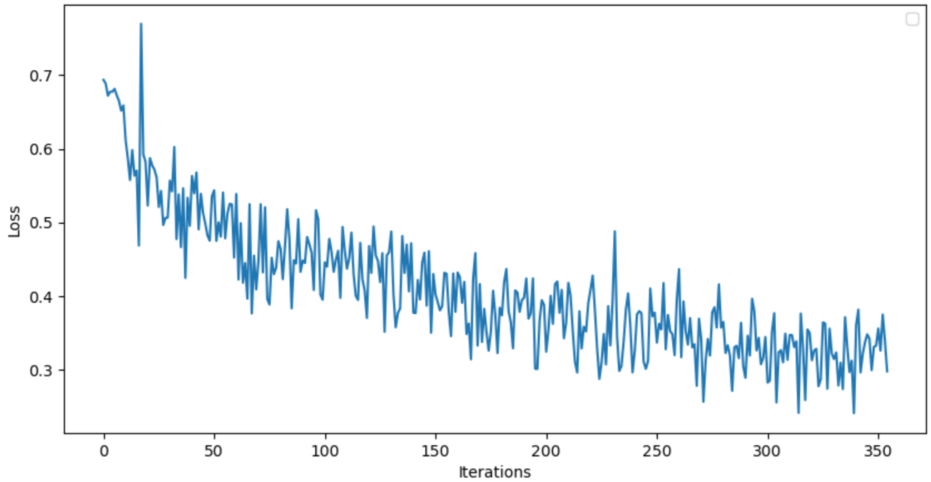

Figure 1: *Loss per iteration Scratch Network*

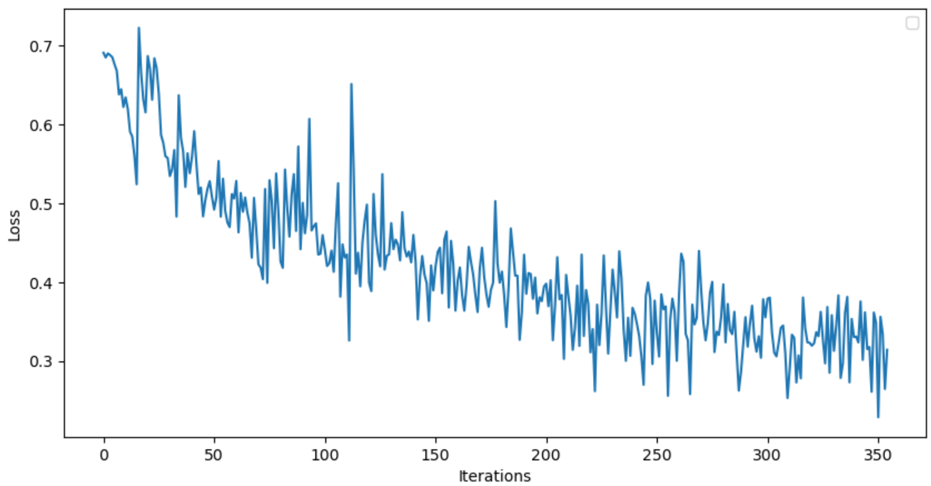

Figure 2: *Loss per iteration No Bidirectional Network*

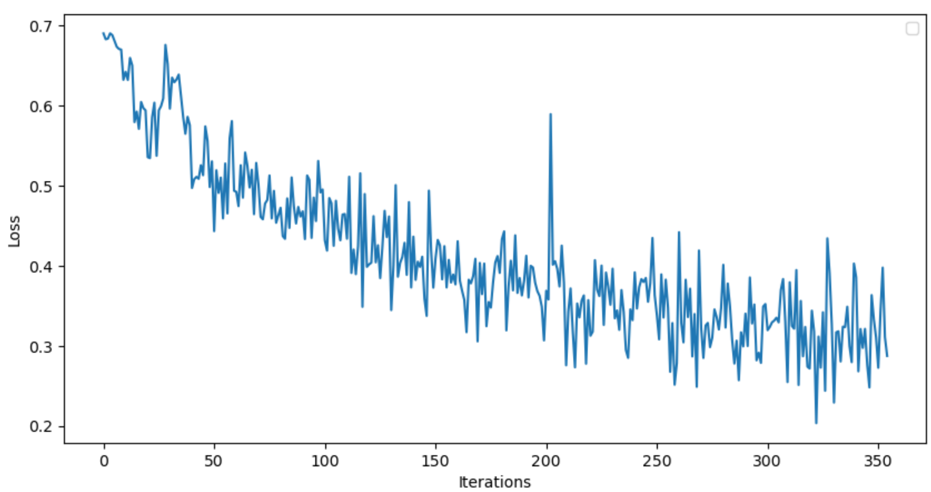

Figure 3: *Loss per iteration Yes Bidirectional Network*

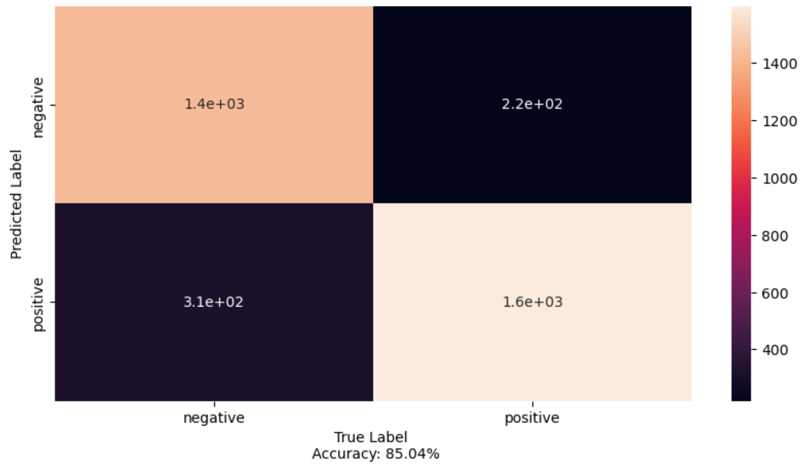

Figure 4: *Confusion Matrix Scratch Network*

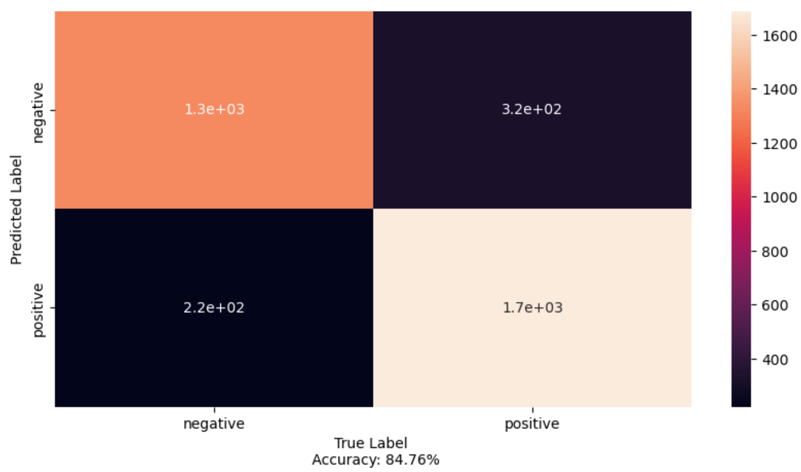

Figure 5: *Confusion Matrix No Bidirectional Network*

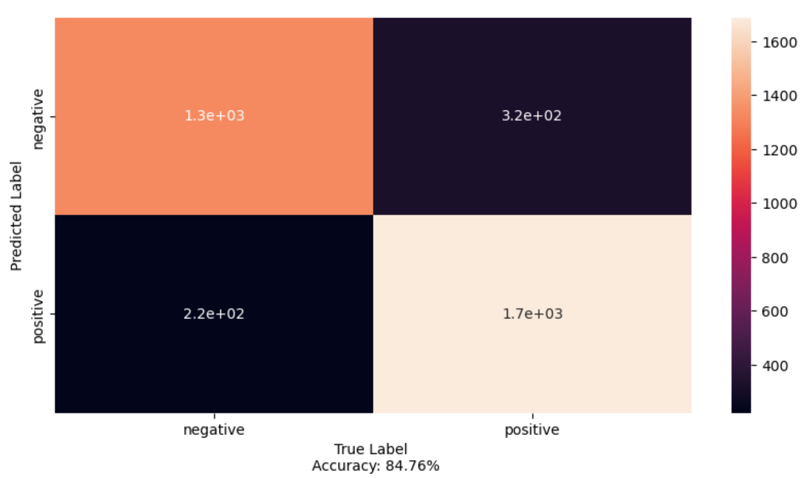

Figure 6: *Confusion Matrix Yes Bidirectional Network*

Overall classification accuracy: 85.04%

	predicted negative	predicted positive
true negative:	86.622%	13.378%
true positive:	16.379%	83.621%

Figure 7: *Accuracy Scratch Network*

Overall classification accuracy: 84.76%

	predicted negative	predicted positive
true negative:	80.508%	19.492%
true positive:	11.617%	88.383%

(a) *Accuracy No Bidirectional Network*

Overall classification accuracy: 84.84%

	predicted negative	predicted positive
true negative:	78.087%	21.913%
true positive:	9.367%	90.633%

Figure 9: *Accuracy Yes Bidirectional Network*

6 Lessons learned

From this assignment, I learned how to apply word embeddings and GRU gates with neural networks.