ECE 60146 Deep Learning **Name:** *Nikita Ravi (ravi30)*
Homework 8
04/17/2023

1 Introduction

The objective of this homework assignment is to gain insight into Recurrent Neural Networks (RNN) and understand how the gating mechanisms in the Gated Recurrent Unit help combat the vanishing gradient problem in RNNs. Finally, we apply these concepts into categorizing text by its sentiment: positive or negative.

2 Theoretical Background

2.1 Word Embeddings

Word embeddings are a fixed-sized numerical representations for words that are learned on the basis of the similarity of word contexts. We use a method called *Word2Vec* is used to obtain the word embeddings. The Word2Vec approach, illustrated in Figure **??**, is as follows:

1. The files in a text corpus are scanned with a window of size $2W + 1$. The word in the middle of the window is considered to be the *focus word* and the W words on either side are known as the *context words* for the *focus word*.
2. The size of the vocabulary is assumed to be V
3. As the text file is scanned, the $V-$element long one-hot vector representation of each focus word is fed as an input to the neural network
4. Each input goes through the first linear layer whose purpose is to be a projection operator where it is multiplied by a matrix $W_{V\times N}$ of learnable parameters. This is done to extract the current value for the embedding for the word and present it to the neural network
5. This projection is then passed into another linear layer with SoftMax as its activation function. This activation function is used because it places equal focus on all the output nodes unlike LogSoftMax which focuses specifically on just the one output node that is supposed to represent the true class label of the input. Using this activation function enables us to talk each output to be the conditional probability of the corresponding word in the vocab being the context word for the input focus word

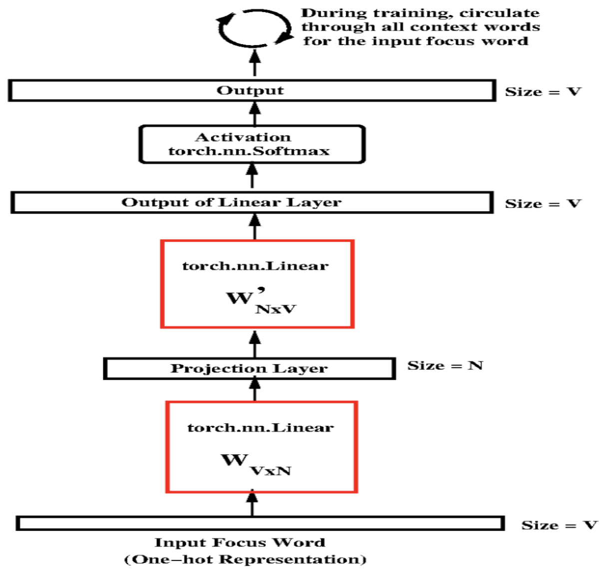


Figure 1: Word2Vec Algorithm

2.2 Recurrent Neural Networks

An effective way to tackle the issue of dealing with variable length input is to use a neural network with feedback called Recurrent Neural Networks (RNN). Feed-forward neural networks in general are meant for data points that are independent of each other. However, if the data points were in a sequence such that one data point was dependent on the previous data points, then the network should incorporate the dependencies between these data points. The benefit of RNNs is that they have the ability to store the states or information of previous inputs to generate the next output of the sequence. The RNN has two inputs: the present and the recent-past (hidden state) and apply weights to both inputs, as shown in figure 2

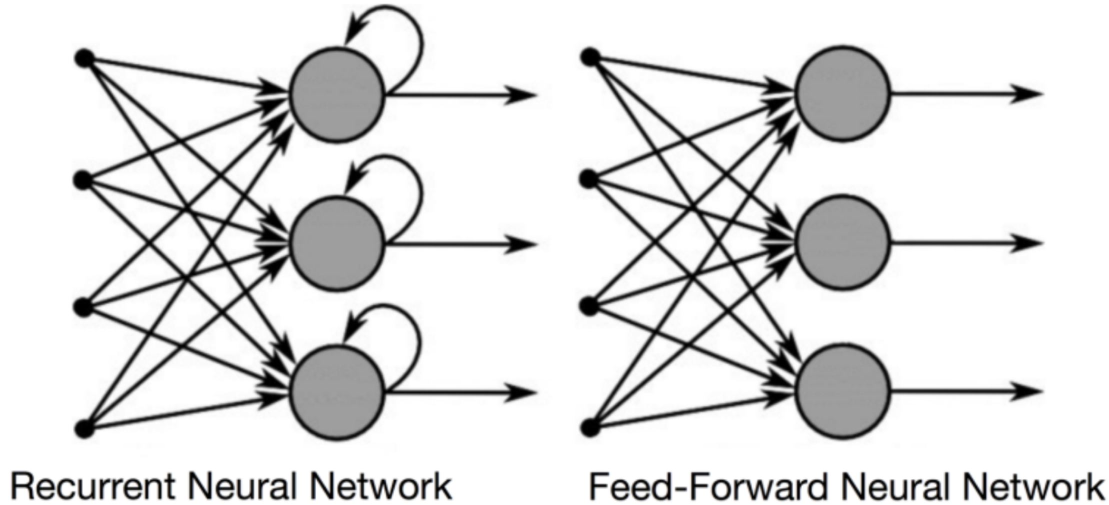


Figure 2: Recurrent Neural Network vs Feed-Forward Neural Network

2.3 Gated Recurrent Unit

The backpropagation of loss in an RNN involves long chain of dependencies because it must span all previous values of the hidden state that contributed to the present value of the output. This results in the short-term dependencies to completely dominate the long-term dependencies leading to the vanishing gradient problem. To deal with this problem we use the Gated Recurrent Unit (GRU) as a solution.

The idea behind a gating mechanism is that we designate a cell to keep the information from the past. Whatever is placed in the cell is subject to being forgotten if it is not relevant to the current state of the input/output relationship. At the same time, the cell can be updated based on the current input/output relationship if that is deemed to be important for future characterizations of the input.

The input to the GRU consists of the ongoing values for the sequences x and h where h represents the hidden state of x. The update gate z as shown in figure 3 helps the model to determine how much of the past information needs to be passed along to the future state. To calculate the update gate, we use the following expression:

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

When the sigmoid function in Equation 1 returns a 0, then the previous value of the hidden state will dominate its current value. On the other hand, when the sigmoid function in Equation 1 returns a 1, then the previous value for the hidden state will be dominated by the $\tilde{h}$ shown in Figure 3. This gate is the reason why we can mitigate the problem of vanishing gradients.

The reset gate is used from the model to decide how much of the past information to forget. To calculate the reset gate, we use the following expression:

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

The reset gate is used to create the *candidate hidden state*, $\tilde{h}$ from the previous state and the input and indicates how much influence the previous hidden state can have on the candidate state. The candidate hidden state is calculated as follows:

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

Finally, the hidden state is updated from its previous hidden state using the expression below

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

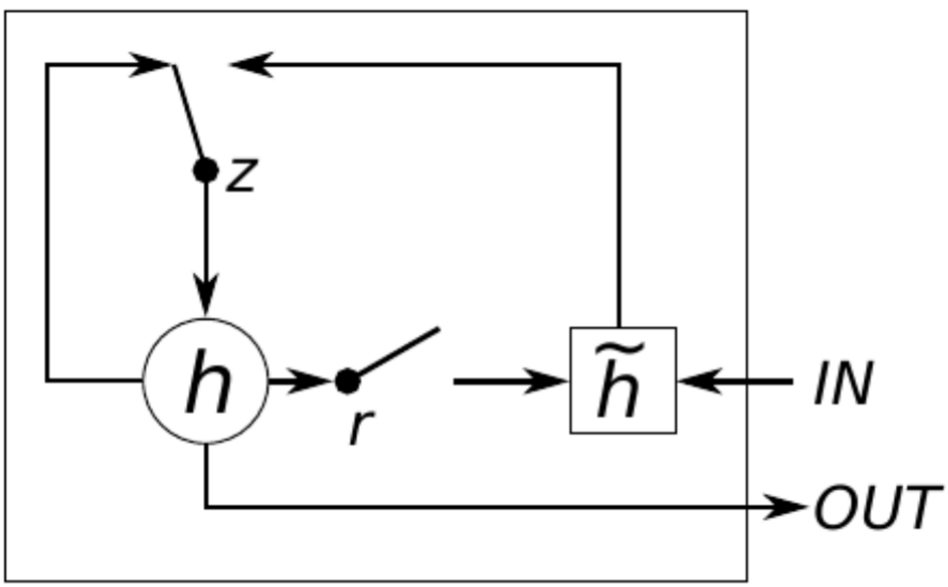


Figure 3: Gated Recurrent Unit

3 Methodology

3.1 Dataloader

The word2vec embedding is obtained from google news and cached after the first time. Each dataset item is saved as a dictionary with the review, the category it belongs to, and finally its ground truth sentiment.

3.2 GRU from Scratch

The equations mentioned above (Eq. 1, 2, 3, 4) are used to create the GRU network from scratch. The parameters W_z, W_r, and W_h, are obtained by passing the input size, which is the size of the word embedding, as the number of input features and the number of output features is set to three times the hidden size, and the output of this linear layer is chunked into three parts along axis one. The same procedure is used to obtain the parameters U_z, U_r, and U_h but this time the number of input features is set to the hidden size. Chunking is a useful method to map the input to N different linear projections thus leading to higher performance. The number of layers indicate the number of GRUs stacked on top of each other. The final hidden state is passed into the LogSoftMax activation function to get the probability that the sequence belongs to a particular sentiment class.

3.3 Training

The input size is the length of the word embedding obtained from Google, and the output size is two because there are only two classes: positive sentiment and negative sentiment.

Parameter	**Value**
Epochs	5
Batch Size	1
Input Size	300
Hidden Size	100
Output Size	2
Learning Rate	1e-4
Betas	(0.9, 0.999)
Optimizer	Adam

Table 1: Hyper-Parameters for Training

4 Implementation and Results

The implementation and the results are shown in the following pages:

Implementation

April 13, 2023

[1]:
```
# Import Libraries
import numpy as np
import torch
import torchvision.transforms as tvt
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import seaborn as sns
import os
from pprint import pprint
import time
import datetime
import gzip
import gensim.downloader as gen_api # free open-source Python library for␣
↪representing documents as semantic vectors
from gensim.models import KeyedVectors
import pickle
import random

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=DeprecationWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

```
/home/dfarache/.conda/envs/cent7/2020.11-py38/eceDL2/lib/python3.8/site-
packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[2]:
```
# Global Variables
train_dir = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw08/
↪sentiment_dataset_train_400.tar.gz"
test_dir = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw08/
↪sentiment_dataset_test_400.tar.gz"
path_to_model = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw08/model"
```

```
path_to_results = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw08/results"
path_to_saved_embeddings = r"/scratch/gilbreth/dfarache/ece60146/Nikita/hw08/
↪word2vec"
batch_size = 1
num_layers = 1
classes = ('negative','positive')

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Torch is on {device}")
device = torch.device(device)
```

```
Torch is on cuda
```

0.1 Generate Datasets

[3]:
```
class GenerateDataset(torch.utils.data.Dataset):
    def __init__(self, data_path, embedding_path):
        super(GenerateDataset, self).__init__()
        self.data_path = data_path
        self.embedding_path = embedding_path

        self.get_word_vectors()
        self.get_sentiment_dataset()

    def get_word_vectors(self):
        if(os.path.exists(os.path.join(self.embedding_path, "vectors.kv"))):
            self.word_vectors = KeyedVectors.load(os.path.join(self.
↪embedding_path, "vectors.kv"))
        else:
            print("Downloading Word2Vec Embeddings")
            self.word_vectors = gen_api.load("word2vec-google-news-300")
            self.word_vectors.save(os.path.join(self.embedding_path, "vectors.
↪kv"))

    def __unzip_datasets(self):
        fptr = gzip.open(self.data_path, mode="rb")
        return fptr.read()

    def __get_indexed_dataset(self):
        self.indexed_dataset = []
        for category in self.positive_reviews:
            for review in self.positive_reviews[category]:
                self.indexed_dataset.append([review, category, 1])

        for category in self.negative_reviews:
            for review in self.negative_reviews[category]:
                self.indexed_dataset.append([review, category, 0])
```

```python
        random.shuffle(self.indexed_dataset)

    def get_sentiment_dataset(self):
        dataset = self.__unzip_datasets()
        self.positive_reviews, self.negative_reviews, self.vocab = pickle.
↪loads(dataset, encoding="latin1")

        self.categories = sorted(list(self.positive_reviews.keys()))
        self.positive_category_frequency = {category: len(self.
↪positive_reviews[category]) for category in self.categories}
        self.negative_category_frequency = {category: len(self.
↪negative_reviews[category]) for category in self.categories}
        self.__get_indexed_dataset()

    def review_to_tensor(self, review):
        list_of_embeddings = []
        for idx, 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.tensor(list_of_embeddings, dtype=torch.float)
        return review_tensor

    def sentiment_to_tensor(self, sentiment):
        sentiment_tensor = torch.zeros(2)
        if(sentiment):
            sentiment_tensor[1] = 1
        else:
            sentiment_tensor[0] = 1
        sentiment_tensor = sentiment_tensor.type(torch.long)
        return sentiment_tensor

    def __len__(self):
        return len(self.indexed_dataset)

    def __getitem__(self, idx):
        sample = self.indexed_dataset[idx]
        review, category, sentiment = sample
        review_tensor = self.review_to_tensor(review) # Shape of review tensor:␣
↪(number of words found in word2vec x length of word2vec)
        sentiment_tensor = self.sentiment_to_tensor(sentiment) # Shape of␣
↪sentiment tensor: (1x2)
        category_idx = self.categories.index(category)
        return {"review": review_tensor, "category": category_idx, "sentiment":␣
↪sentiment_tensor}
```

[4]:
```
def generate_dataloader(data_path, embedding_path, debug=False):
    dataset = GenerateDataset(data_path, embedding_path)
    if(debug):
        pprint(dataset[2])
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,␣
↪shuffle=True, num_workers=2, drop_last=True)
    return dataloader

trainloader = generate_dataloader(train_dir, path_to_saved_embeddings,␣
↪debug=True)
testloader = generate_dataloader(test_dir, path_to_saved_embeddings)
```

```
{'category': 13,
 'review': tensor([[-0.2256, -0.0195,  0.0908,  …,  0.0282, -0.1777, -0.0060],
        [ 0.0571, -0.0527, -0.1172,  …, -0.0444,  0.0138, -0.0381],
        [ 0.1094,  0.1406, -0.0317,  …,  0.0077,  0.1201, -0.1797],
        …,
        [ 0.0081,  0.2578,  0.2471,  …, -0.3359,  0.0322, -0.0698],
        [ 0.0801,  0.1050,  0.0498,  …,  0.0037,  0.0476, -0.0688],
        [ 0.1416, -0.0271, -0.1846,  …,  0.0143,  0.1484, -0.0383]]),
 'sentiment': tensor([0, 1])}
```

0.2 Network

0.2.1 Task 1

[5]:
```
class GRUCell(nn.Module):
    # Inspired by https://github.com/georgeyiasemis/
↪Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py
    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.Z = nn.Linear(in_features=self.input_size, out_features=3*self.
↪hidden_size, bias=self.bias)
        self.U = nn.Linear(in_features=self.hidden_size, out_features=3*self.
↪hidden_size, bias=self.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, X, hidden_state=None):
        if(hidden_state is None):
            hidden_state = torch.zeros((batch_size, self.hidden_size),␣
↪device=device, dtype=X.dtype, requires_grad=True)

        Z_t = self.Z(X)
        U_t = self.U(hidden_state)

        """
        Similar to creating N nn.Linear layers and doing forward pass with all␣
↪N of them, we can instead
        create a single linear layer, do one forward pass and just chunk the␣
↪output into N pieces
        """

        Wzx, Wrx, Whx = Z_t.chunk(3, dim=1)
        Uzx, Urx, Uhx = U_t.chunk(3, dim=1)

        reset_gate = F.sigmoid(Wrx + Urx)
        update_gate = F.sigmoid(Wzx + Uzx)
        candidate_hidden_gate = F.tanh(Whx + (reset_gate * Uhx))

        hidden_state = update_gate * hidden_state + (1 - update_gate) *␣
↪candidate_hidden_gate
        return hidden_state
```

[6]:
```
class GRU(nn.Module):
    # Inspired by https://github.com/georgeyiasemis/
↪Recurrent-Neural-Networks-from-scratch-using-PyTorch/blob/main/rnnmodels.py
    def __init__(self, input_size, hidden_size, output_size, num_layers, bias):
        super(GRU, self).__init__()
        self.input_size = input_size # Size of the tensor for each word in a␣
↪sequence of words. Since we are using the word2vec embedding, the value of␣
↪this variable will always be 300
        self.hidden_size = hidden_size # Size of the hidden state in the RNN
        self.output_size = output_size # Output of the RNN, in this case will␣
↪be 2 (positive, negative)
        self.num_layers = num_layers # Create a stack of GRUs
        self.bias = bias

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

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

        for seq in range(inputs.shape[1]):
            for layer in range(self.num_layers):
                if(not layer):
                    hidden_layer = self.rnn_cell_list[layer](inputs[:, seq, :],␣
 ↪hidden[layer])
                else:
                    hidden_layer = self.rnn_cell_list[layer](hidden[layer - 1],␣
 ↪hidden[layer])
                hidden[layer] = hidden_layer

            outputs.append(hidden_layer)

        final_hidden_layer = outputs[-1]
        final_hidden_layer = self.fc(final_hidden_layer)
        final_hidden_layer = self.logSoftMax(final_hidden_layer)
        return final_hidden_layer
```

[7]:
```
# Number of layers and learnable parameters in the GRU Network
model = GRU(input_size=300, hidden_size=100, output_size=2,␣
 ↪num_layers=num_layers, bias=True)
num_layers_in_model = len(list(model.parameters()))
num_learnable_parameters = sum(p.numel() for p in model.parameters() if p.
 ↪requires_grad)

print(f"Number of layers in the GRU with Embeddings network:␣
 ↪{num_layers_in_model}")
print(f"Number of learnable parameters in the GRU with Embeddings network:␣
 ↪{num_learnable_parameters}")
```

```
Number of layers in the GRU with Embeddings network: 6
Number of learnable parameters in the GRU with Embeddings network: 120802
```

0.2.2 Task 2

[8]:
```
class GRUnetWithEmbedding(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1,␣
 ↪bidirectional=False):
        # Inspired by Professor Kak's GRUnetWithEmbedding
        super(GRUnetWithEmbedding, self).__init__()
        self.input_size = input_size # Size of the tensor for each word in a␣
 ↪sequence of words. Since we are using the word2vec embedding, the value of␣
 ↪this variable will always be 300
        self.hidden_size = hidden_size # Size of the hidden state in the RNN
        self.output_size = output_size # Output of the RNN, in this case will␣
 ↪be 2 (positive, negative)
        self.num_layers = num_layers # Create a stack of GRUs
        self.bidirectional = bidirectional
        self.gru = nn.GRU(input_size=self.input_size, hidden_size=self.
 ↪hidden_size, num_layers=self.num_layers, bidirectional=self.bidirectional,␣
 ↪batch_first=True)

        self.num_bidirectional = 2 if self.bidirectional else 1
        self.fc = nn.Linear(in_features=self.hidden_size * self.num_layers *␣
 ↪self.num_bidirectional, out_features=self.output_size)
        self.logSoftMax = nn.LogSoftmax(dim=1)

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

    def init_hidden(self):
        weight = next(self.parameters()).data
        hidden = weight.new_zeros((num_layers * self.num_bidirectional,␣
 ↪batch_size, self.hidden_size)) # create a new tensor of the same datatype␣
 ↪filled with zeros - useful when we don't know what the priori datatype is
        return hidden
```

[9]:
```
# Number of layers and learnable parameters in the GRUnetWithEmbedding Network
model = GRUnetWithEmbedding(input_size=300, hidden_size=100, output_size=2,␣
 ↪num_layers=num_layers)
num_layers_in_model = len(list(model.parameters()))
num_learnable_parameters = sum(p.numel() for p in model.parameters() if p.
 ↪requires_grad)

print(f"Number of layers in the GRU with Embeddings network:␣
 ↪{num_layers_in_model}")
```

```
print(f"Number of learnable parameters in the GRU with Embeddings network:␣
↪{num_learnable_parameters}")
```

```
Number of layers in the GRU with Embeddings network: 6
Number of learnable parameters in the GRU with Embeddings network: 120802
```

0.3 Training and Testing

[10]:
```
def plot_losses(loss, epochs, mode="scratch"):
    # Plot the training losses
    iterations = range(len(loss))
    plt.plot(iterations, loss)

    plt.title("Training Loss")
    plt.xlabel(f"Iterations over {epochs} epochs")
    plt.ylabel("Loss")
    plt.legend(loc ="upper right")

    filename = "train_loss_" + mode + ".jpg"
    plt.savefig(os.path.join(path_to_results, filename))
    plt.show()
```

[11]:
```
def train(trainloader, criterion, lr, betas, epochs, log=200, task=1,␣
↪bidirectional=False):

    if(task==1):
        net = GRU(input_size=300, hidden_size=100, output_size=2, num_layers=1,␣
↪bias=True)
    elif(task==2):
        net = GRUnetWithEmbedding(input_size=300, hidden_size=100,␣
↪output_size=2, num_layers=num_layers, bidirectional=bidirectional)


    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas) # Adam␣
↪Optimizer
    net = net.to(device)
    training_loss = []

    print(f"Training started at time {datetime.datetime.now().time()}")
    start_time = time.time()
    check_loss = float("inf")

    for epoch in range(1, epochs + 1):
        running_loss = 0.0
        for batch_idx, data in enumerate(trainloader):
            review, category, sentiment = data["review"], data["category"],␣
↪data["sentiment"]
```

```
            review = review.to(device)
            category = category.to(device)
            sentiment = sentiment.to(device)

            optimizer.zero_grad()
            if(task==1):
                output = net(review)
            elif(task==2):
                output, hidden = net(review, net.init_hidden().to(device))

            loss = criterion(output, torch.argmax(sentiment, dim=1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()

            if batch_idx % log == log-1:
                average_loss = running_loss / float(log)
                training_loss.append(average_loss)
                current_time = time.time()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time:%4d secs]     loss: %.
↪5f" % (epoch, batch_idx+1, time_elapsed, average_loss))

                if(running_loss < check_loss):
                    check_loss = running_loss
                    bidirectional_name = "bid" if bidirectional else "no_bid"
                    torch.save(net.state_dict(), os.path.join(path_to_model,␣
↪"Task " + str(task) + "_" + bidirectional_name + ".pt"))

                running_loss = 0.0

    print("Lowest loss achieved by network: %.4f" % (check_loss / float(log)))
    print("Training finished in %4d secs" % (time.time() - start_time))
    return net, training_loss
```

[12]:
```
def test(net, testloader, log=100, task=1, bidirectional="no_bid"):
    model_name = "Task " + str(task) + "_" + bidirectional + ".pt"
    net.load_state_dict(torch.load(os.path.join(path_to_model, model_name)))
    net = net.to(device)
    confusion_matrix = np.zeros((len(classes), len(classes)))

    with torch.no_grad():
        for batch_idx, data in enumerate(testloader):
            review, category, sentiment = data["review"], data["category"],␣
↪data["sentiment"]
            review = review.to(device)
```

```
            category = category.to(device)
            sentiment = sentiment.to(device)

            if(task==1):
                output = net(review)
            elif(task==2):
                output, hidden = net(review, net.init_hidden().to(device))

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

            if(batch_idx % log == log - 1):
                print("    [batch_idx=%d]     predicted_label=%d      ␣
 ↪gt_label=%d" % (batch_idx+1, predicted_idx, gt_idx))
            confusion_matrix[gt_idx, predicted_idx] += 1

    accuracy = np.trace(confusion_matrix) / np.sum(confusion_matrix)
    return confusion_matrix, accuracy
```

[13]:
```
def display_confusion_matrix(conf, accuracy, class_list=classes, task=1,␣
 ↪bidirectional="no_bid"):
    figure = plt.figure(3)
    sns.heatmap(conf, xticklabels=class_list, yticklabels=class_list,␣
 ↪annot=True)
    plt.xlabel(f"True Label \n Accuracy: {accuracy}")
    plt.ylabel("Predicted Label")

    filename = "conf_" + str(task) + "_" + bidirectional + ".jpg"
    plt.savefig(os.path.join(path_to_results, filename))
```

0.4 Task 1

0.4.1 Training

[14]:
```
# Parameters for training
lr = 1e-4 # Learning Rate
betas = (0.9, 0.999) # Betas factor
epochs = 5 # Number of epochs to train
criterion = nn.NLLLoss() # Negative Log Likelihood Loss
```

[15]:
```
net1, training_loss = train(trainloader, criterion, lr, betas, epochs, log=800,␣
 ↪task=1)
plot_losses(training_loss, epochs, mode="scratch")
```

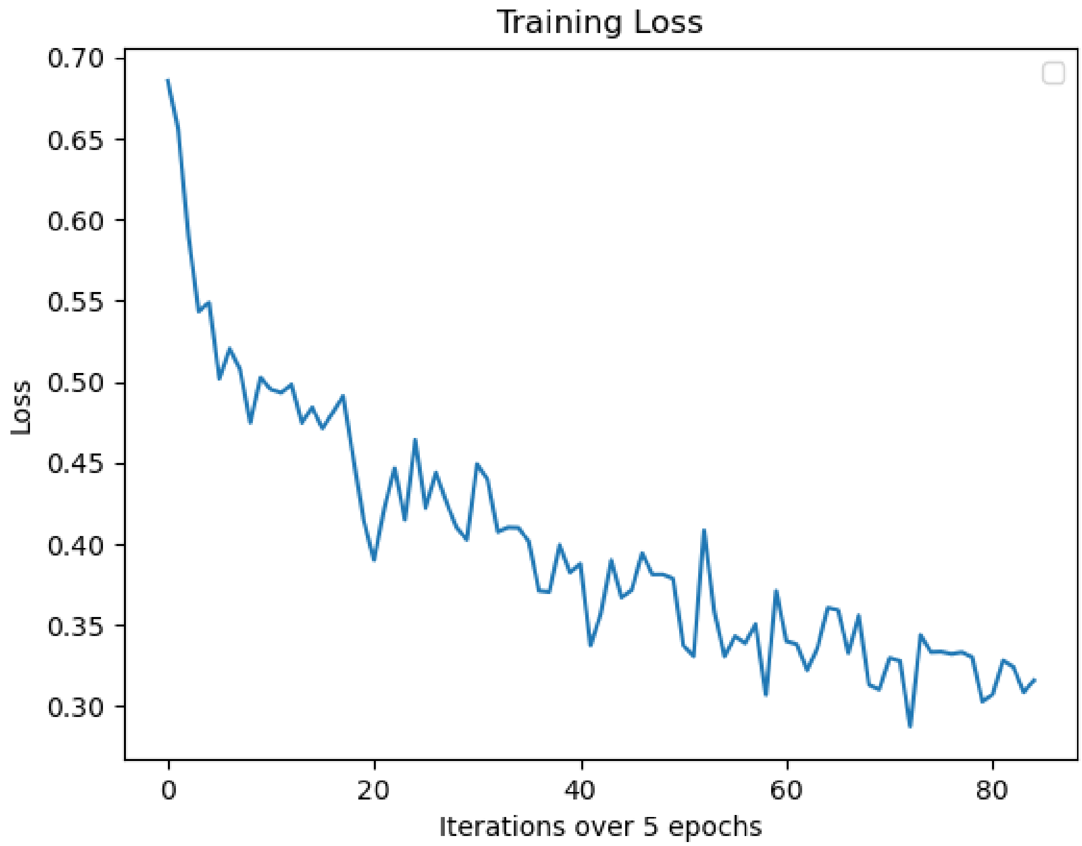
```
Training started at time 19:13:21.302765
[epoch:1  iter: 800  elapsed_time:  92 secs]     loss: 0.68544
[epoch:1  iter:1600  elapsed_time: 180 secs]     loss: 0.65556
```

```
[epoch:1  iter:2400  elapsed_time: 269 secs]     loss: 0.58972
[epoch:1  iter:3200  elapsed_time: 362 secs]     loss: 0.54331
[epoch:1  iter:4000  elapsed_time: 453 secs]     loss: 0.54876
[epoch:1  iter:4800  elapsed_time: 546 secs]     loss: 0.50168
[epoch:1  iter:5600  elapsed_time: 639 secs]     loss: 0.52044
[epoch:1  iter:6400  elapsed_time: 733 secs]     loss: 0.50741
[epoch:1  iter:7200  elapsed_time: 824 secs]     loss: 0.47482
[epoch:1  iter:8000  elapsed_time: 918 secs]     loss: 0.50240
[epoch:1  iter:8800  elapsed_time:1010 secs]     loss: 0.49524
[epoch:1  iter:9600  elapsed_time:1101 secs]     loss: 0.49316
[epoch:1  iter:10400  elapsed_time:1187 secs]     loss: 0.49830
[epoch:1  iter:11200  elapsed_time:1284 secs]     loss: 0.47464
[epoch:1  iter:12000  elapsed_time:1377 secs]     loss: 0.48407
[epoch:1  iter:12800  elapsed_time:1466 secs]     loss: 0.47121
[epoch:1  iter:13600  elapsed_time:1564 secs]     loss: 0.48095
[epoch:2  iter: 800  elapsed_time:1728 secs]     loss: 0.49112
[epoch:2  iter:1600  elapsed_time:1820 secs]     loss: 0.45282
[epoch:2  iter:2400  elapsed_time:1909 secs]     loss: 0.41447
[epoch:2  iter:3200  elapsed_time:1997 secs]     loss: 0.38995
[epoch:2  iter:4000  elapsed_time:2088 secs]     loss: 0.42206
[epoch:2  iter:4800  elapsed_time:2179 secs]     loss: 0.44663
[epoch:2  iter:5600  elapsed_time:2269 secs]     loss: 0.41500
[epoch:2  iter:6400  elapsed_time:2363 secs]     loss: 0.46402
[epoch:2  iter:7200  elapsed_time:2453 secs]     loss: 0.42213
[epoch:2  iter:8000  elapsed_time:2548 secs]     loss: 0.44407
[epoch:2  iter:8800  elapsed_time:2638 secs]     loss: 0.42608
[epoch:2  iter:9600  elapsed_time:2722 secs]     loss: 0.41016
[epoch:2  iter:10400  elapsed_time:2812 secs]     loss: 0.40253
[epoch:2  iter:11200  elapsed_time:2905 secs]     loss: 0.44917
[epoch:2  iter:12000  elapsed_time:2996 secs]     loss: 0.43971
[epoch:2  iter:12800  elapsed_time:3090 secs]     loss: 0.40745
[epoch:2  iter:13600  elapsed_time:3182 secs]     loss: 0.41014
[epoch:3  iter: 800  elapsed_time:3347 secs]     loss: 0.40993
[epoch:3  iter:1600  elapsed_time:3443 secs]     loss: 0.40159
[epoch:3  iter:2400  elapsed_time:3530 secs]     loss: 0.37119
[epoch:3  iter:3200  elapsed_time:3621 secs]     loss: 0.37041
[epoch:3  iter:4000  elapsed_time:3714 secs]     loss: 0.39950
[epoch:3  iter:4800  elapsed_time:3805 secs]     loss: 0.38240
[epoch:3  iter:5600  elapsed_time:3906 secs]     loss: 0.38768
[epoch:3  iter:6400  elapsed_time:3993 secs]     loss: 0.33749
[epoch:3  iter:7200  elapsed_time:4082 secs]     loss: 0.35709
[epoch:3  iter:8000  elapsed_time:4164 secs]     loss: 0.39021
[epoch:3  iter:8800  elapsed_time:4256 secs]     loss: 0.36685
[epoch:3  iter:9600  elapsed_time:4344 secs]     loss: 0.37175
[epoch:3  iter:10400  elapsed_time:4437 secs]     loss: 0.39426
[epoch:3  iter:11200  elapsed_time:4524 secs]     loss: 0.38121
[epoch:3  iter:12000  elapsed_time:4607 secs]     loss: 0.38127
[epoch:3  iter:12800  elapsed_time:4688 secs]     loss: 0.37854
```

```
[epoch:3  iter:13600  elapsed_time:4774 secs]     loss: 0.33746
[epoch:4  iter: 800  elapsed_time:4935 secs]     loss: 0.33084
[epoch:4  iter:1600  elapsed_time:5022 secs]     loss: 0.40831
[epoch:4  iter:2400  elapsed_time:5110 secs]     loss: 0.35834
[epoch:4  iter:3200  elapsed_time:5192 secs]     loss: 0.33079
[epoch:4  iter:4000  elapsed_time:5278 secs]     loss: 0.34311
[epoch:4  iter:4800  elapsed_time:5361 secs]     loss: 0.33881
[epoch:4  iter:5600  elapsed_time:5446 secs]     loss: 0.35059
[epoch:4  iter:6400  elapsed_time:5532 secs]     loss: 0.30697
[epoch:4  iter:7200  elapsed_time:5616 secs]     loss: 0.37102
[epoch:4  iter:8000  elapsed_time:5704 secs]     loss: 0.34012
[epoch:4  iter:8800  elapsed_time:5794 secs]     loss: 0.33819
[epoch:4  iter:9600  elapsed_time:5884 secs]     loss: 0.32210
[epoch:4  iter:10400  elapsed_time:5974 secs]     loss: 0.33584
[epoch:4  iter:11200  elapsed_time:6067 secs]     loss: 0.36070
[epoch:4  iter:12000  elapsed_time:6155 secs]     loss: 0.35931
[epoch:4  iter:12800  elapsed_time:6245 secs]     loss: 0.33282
[epoch:4  iter:13600  elapsed_time:6338 secs]     loss: 0.35593
[epoch:5  iter: 800  elapsed_time:6497 secs]     loss: 0.31321
[epoch:5  iter:1600  elapsed_time:6588 secs]     loss: 0.31026
[epoch:5  iter:2400  elapsed_time:6685 secs]     loss: 0.32981
[epoch:5  iter:3200  elapsed_time:6780 secs]     loss: 0.32789
[epoch:5  iter:4000  elapsed_time:6877 secs]     loss: 0.28764
[epoch:5  iter:4800  elapsed_time:6975 secs]     loss: 0.34385
[epoch:5  iter:5600  elapsed_time:7074 secs]     loss: 0.33338
[epoch:5  iter:6400  elapsed_time:7167 secs]     loss: 0.33362
[epoch:5  iter:7200  elapsed_time:7261 secs]     loss: 0.33226
[epoch:5  iter:8000  elapsed_time:7355 secs]     loss: 0.33326
[epoch:5  iter:8800  elapsed_time:7446 secs]     loss: 0.33039
[epoch:5  iter:9600  elapsed_time:7543 secs]     loss: 0.30293
[epoch:5  iter:10400  elapsed_time:7640 secs]     loss: 0.30730
[epoch:5  iter:11200  elapsed_time:7729 secs]     loss: 0.32823
[epoch:5  iter:12000  elapsed_time:7826 secs]     loss: 0.32422
[epoch:5  iter:12800  elapsed_time:7914 secs]     loss: 0.30868
[epoch:5  iter:13600  elapsed_time:8006 secs]     loss: 0.31597
```

```
No handles with labels found to put in legend.
```

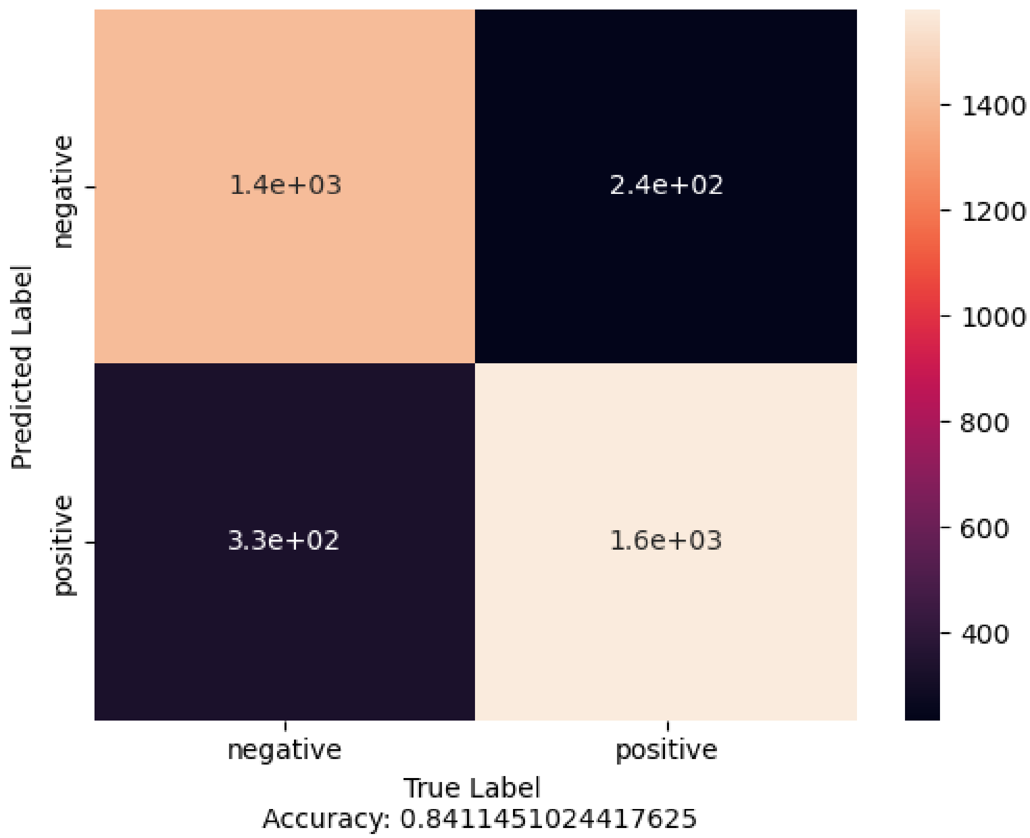
```
Lowest loss achieved by network: 0.2876
Training finished in 8075 secs
```



0.4.2 Testing

[16]:
```
confusion_matrix, accuracy = test(net1, testloader, log=500, task=1,␣
  ↪bidirectional="no_bid")
display_confusion_matrix(confusion_matrix, accuracy, task=1)
```

```
[batch_idx=500]     predicted_label=1       gt_label=1
[batch_idx=1000]     predicted_label=1       gt_label=1
[batch_idx=1500]     predicted_label=1       gt_label=1
[batch_idx=2000]     predicted_label=0       gt_label=0
[batch_idx=2500]     predicted_label=1       gt_label=1
[batch_idx=3000]     predicted_label=0       gt_label=0
[batch_idx=3500]     predicted_label=1       gt_label=0
```



0.5 Task 2

0.5.1 Without Bidirectional

Training

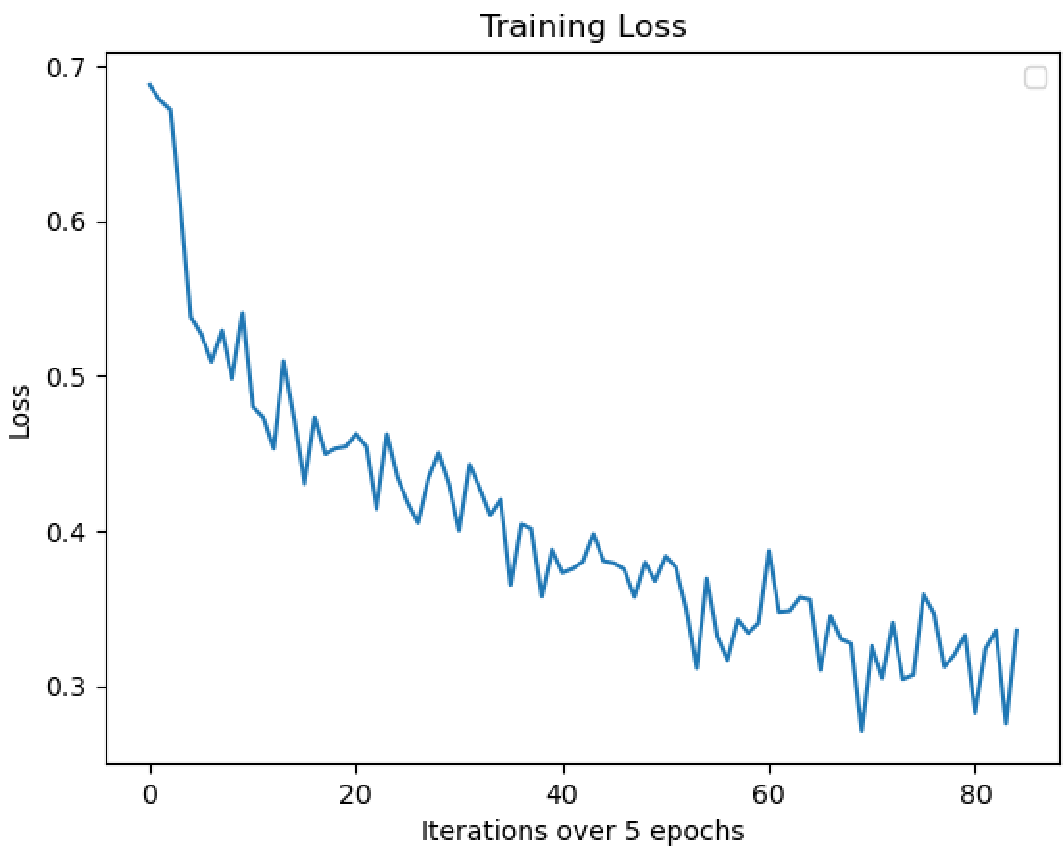
[17]:
```
# Parameters for training
lr = 1e-4 # Learning Rate
betas = (0.9, 0.999) # Betas factor
epochs = 5 # Number of epochs to train
criterion = nn.NLLLoss() # Negative Log Likelihood Loss
```

[18]:
```
net2, training_loss = train(trainloader, criterion, lr, betas, epochs, log=800,␣
  ↪task=2, bidirectional=False)
plot_losses(training_loss, epochs, mode="torch_nobid")
```

```
Training started at time 21:29:15.033771
[epoch:1  iter: 800  elapsed_time:   4 secs]     loss: 0.68759
[epoch:1  iter:1600  elapsed_time:   8 secs]     loss: 0.67777
[epoch:1  iter:2400  elapsed_time:  12 secs]     loss: 0.67158
[epoch:1  iter:3200  elapsed_time:  17 secs]     loss: 0.60928
[epoch:1  iter:4000  elapsed_time:  21 secs]     loss: 0.53789
```

```
[epoch:1  iter:4800  elapsed_time:  25 secs]     loss: 0.52661
[epoch:1  iter:5600  elapsed_time:  30 secs]     loss: 0.50897
[epoch:1  iter:6400  elapsed_time:  34 secs]     loss: 0.52906
[epoch:1  iter:7200  elapsed_time:  38 secs]     loss: 0.49797
[epoch:1  iter:8000  elapsed_time:  42 secs]     loss: 0.54044
[epoch:1  iter:8800  elapsed_time:  46 secs]     loss: 0.48009
[epoch:1  iter:9600  elapsed_time:  51 secs]     loss: 0.47346
[epoch:1  iter:10400  elapsed_time:  56 secs]     loss: 0.45292
[epoch:1  iter:11200  elapsed_time:  60 secs]     loss: 0.50940
[epoch:1  iter:12000  elapsed_time:  65 secs]     loss: 0.47234
[epoch:1  iter:12800  elapsed_time:  70 secs]     loss: 0.43019
[epoch:1  iter:13600  elapsed_time:  74 secs]     loss: 0.47327
[epoch:2  iter: 800  elapsed_time:  82 secs]     loss: 0.44941
[epoch:2  iter:1600  elapsed_time:  86 secs]     loss: 0.45295
[epoch:2  iter:2400  elapsed_time:  91 secs]     loss: 0.45423
[epoch:2  iter:3200  elapsed_time:  95 secs]     loss: 0.46247
[epoch:2  iter:4000  elapsed_time:  99 secs]     loss: 0.45418
[epoch:2  iter:4800  elapsed_time: 103 secs]     loss: 0.41401
[epoch:2  iter:5600  elapsed_time: 107 secs]     loss: 0.46221
[epoch:2  iter:6400  elapsed_time: 112 secs]     loss: 0.43455
[epoch:2  iter:7200  elapsed_time: 116 secs]     loss: 0.41842
[epoch:2  iter:8000  elapsed_time: 120 secs]     loss: 0.40504
[epoch:2  iter:8800  elapsed_time: 124 secs]     loss: 0.43328
[epoch:2  iter:9600  elapsed_time: 129 secs]     loss: 0.45014
[epoch:2  iter:10400  elapsed_time: 133 secs]     loss: 0.42981
[epoch:2  iter:11200  elapsed_time: 138 secs]     loss: 0.40008
[epoch:2  iter:12000  elapsed_time: 142 secs]     loss: 0.44270
[epoch:2  iter:12800  elapsed_time: 146 secs]     loss: 0.42749
[epoch:2  iter:13600  elapsed_time: 151 secs]     loss: 0.41002
[epoch:3  iter: 800  elapsed_time: 159 secs]     loss: 0.42005
[epoch:3  iter:1600  elapsed_time: 164 secs]     loss: 0.36506
[epoch:3  iter:2400  elapsed_time: 168 secs]     loss: 0.40417
[epoch:3  iter:3200  elapsed_time: 173 secs]     loss: 0.40128
[epoch:3  iter:4000  elapsed_time: 177 secs]     loss: 0.35749
[epoch:3  iter:4800  elapsed_time: 182 secs]     loss: 0.38775
[epoch:3  iter:5600  elapsed_time: 186 secs]     loss: 0.37287
[epoch:3  iter:6400  elapsed_time: 190 secs]     loss: 0.37557
[epoch:3  iter:7200  elapsed_time: 195 secs]     loss: 0.38001
[epoch:3  iter:8000  elapsed_time: 199 secs]     loss: 0.39787
[epoch:3  iter:8800  elapsed_time: 203 secs]     loss: 0.38049
[epoch:3  iter:9600  elapsed_time: 208 secs]     loss: 0.37899
[epoch:3  iter:10400  elapsed_time: 212 secs]     loss: 0.37494
[epoch:3  iter:11200  elapsed_time: 217 secs]     loss: 0.35724
[epoch:3  iter:12000  elapsed_time: 222 secs]     loss: 0.37960
[epoch:3  iter:12800  elapsed_time: 226 secs]     loss: 0.36755
[epoch:3  iter:13600  elapsed_time: 230 secs]     loss: 0.38343
[epoch:4  iter: 800  elapsed_time: 238 secs]     loss: 0.37640
[epoch:4  iter:1600  elapsed_time: 243 secs]     loss: 0.34966
```

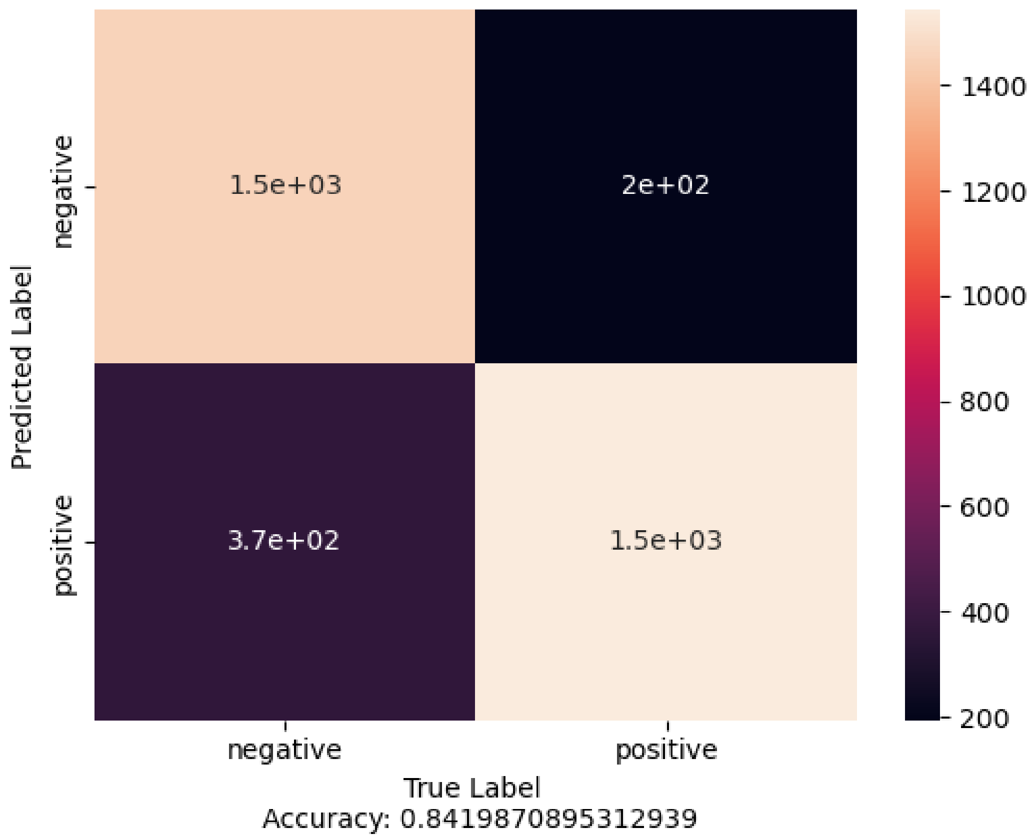
```
[epoch:4  iter:2400  elapsed_time: 247 secs]     loss: 0.31125
[epoch:4  iter:3200  elapsed_time: 252 secs]     loss: 0.36918
[epoch:4  iter:4000  elapsed_time: 256 secs]     loss: 0.33169
[epoch:4  iter:4800  elapsed_time: 260 secs]     loss: 0.31639
[epoch:4  iter:5600  elapsed_time: 265 secs]     loss: 0.34248
[epoch:4  iter:6400  elapsed_time: 269 secs]     loss: 0.33395
[epoch:4  iter:7200  elapsed_time: 274 secs]     loss: 0.34016
[epoch:4  iter:8000  elapsed_time: 278 secs]     loss: 0.38690
[epoch:4  iter:8800  elapsed_time: 282 secs]     loss: 0.34758
[epoch:4  iter:9600  elapsed_time: 287 secs]     loss: 0.34830
[epoch:4  iter:10400  elapsed_time: 291 secs]     loss: 0.35708
[epoch:4  iter:11200  elapsed_time: 296 secs]     loss: 0.35550
[epoch:4  iter:12000  elapsed_time: 300 secs]     loss: 0.31010
[epoch:4  iter:12800  elapsed_time: 304 secs]     loss: 0.34500
[epoch:4  iter:13600  elapsed_time: 309 secs]     loss: 0.32983
[epoch:5  iter: 800  elapsed_time: 317 secs]     loss: 0.32710
[epoch:5  iter:1600  elapsed_time: 321 secs]     loss: 0.27101
[epoch:5  iter:2400  elapsed_time: 325 secs]     loss: 0.32566
[epoch:5  iter:3200  elapsed_time: 329 secs]     loss: 0.30487
[epoch:5  iter:4000  elapsed_time: 333 secs]     loss: 0.34052
[epoch:5  iter:4800  elapsed_time: 338 secs]     loss: 0.30414
[epoch:5  iter:5600  elapsed_time: 342 secs]     loss: 0.30700
[epoch:5  iter:6400  elapsed_time: 347 secs]     loss: 0.35900
[epoch:5  iter:7200  elapsed_time: 351 secs]     loss: 0.34695
[epoch:5  iter:8000  elapsed_time: 355 secs]     loss: 0.31206
[epoch:5  iter:8800  elapsed_time: 359 secs]     loss: 0.32000
[epoch:5  iter:9600  elapsed_time: 363 secs]     loss: 0.33257
[epoch:5  iter:10400  elapsed_time: 368 secs]     loss: 0.28222
[epoch:5  iter:11200  elapsed_time: 372 secs]     loss: 0.32379
[epoch:5  iter:12000  elapsed_time: 377 secs]     loss: 0.33567
[epoch:5  iter:12800  elapsed_time: 381 secs]     loss: 0.27562
[epoch:5  iter:13600  elapsed_time: 385 secs]     loss: 0.33557

No handles with labels found to put in legend.

Lowest loss achieved by network: 0.2710
Training finished in  388 secs
```



Testing

[19]:
```
confusion_matrix, accuracy = test(net2, testloader, log=500, task=2,␣
  ↪bidirectional="no_bid")
display_confusion_matrix(confusion_matrix, accuracy, task=2)
```

```
[batch_idx=500]     predicted_label=1        gt_label=1
[batch_idx=1000]     predicted_label=1        gt_label=1
[batch_idx=1500]     predicted_label=0        gt_label=0
[batch_idx=2000]     predicted_label=0        gt_label=0
[batch_idx=2500]     predicted_label=0        gt_label=0
[batch_idx=3000]     predicted_label=0        gt_label=0
[batch_idx=3500]     predicted_label=0        gt_label=1
```



0.5.2 With Bidirectional

Training

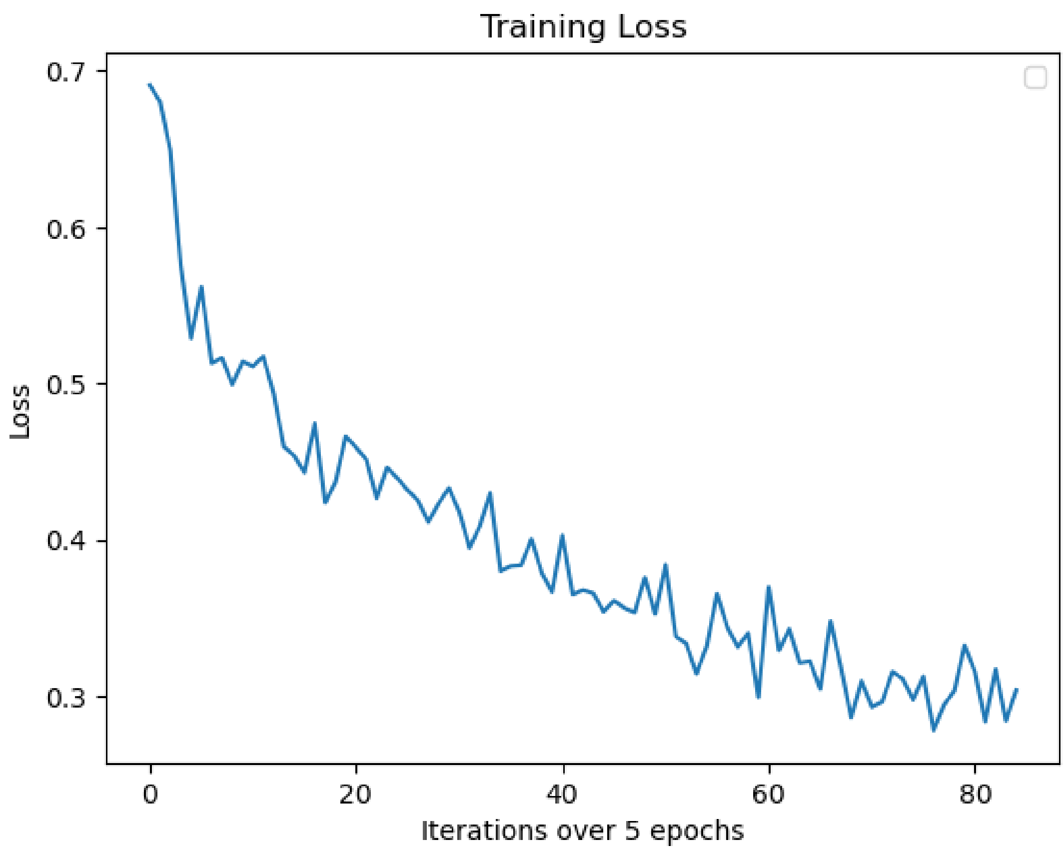
[20]:
```
# Parameters for training
lr = 1e-4 # Learning Rate
betas = (0.9, 0.999) # Betas factor
epochs = 5 # Number of epochs to train
criterion = nn.NLLLoss() # Negative Log Likelihood Loss
```

[21]:
```
net2, training_loss = train(trainloader, criterion, lr, betas, epochs, log=800,␣
  ↪task=2, bidirectional=True)
plot_losses(training_loss, epochs, mode="torch_bid")
```

```
Training started at time 21:35:58.780608
[epoch:1  iter: 800  elapsed_time:   5 secs]     loss: 0.69086
[epoch:1  iter:1600  elapsed_time:  10 secs]     loss: 0.68022
[epoch:1  iter:2400  elapsed_time:  16 secs]     loss: 0.64880
[epoch:1  iter:3200  elapsed_time:  22 secs]     loss: 0.57599
[epoch:1  iter:4000  elapsed_time:  27 secs]     loss: 0.52899
[epoch:1  iter:4800  elapsed_time:  32 secs]     loss: 0.56169
```

```
[epoch:1  iter:5600  elapsed_time:  38 secs]     loss: 0.51305
[epoch:1  iter:6400  elapsed_time:  43 secs]     loss: 0.51645
[epoch:1  iter:7200  elapsed_time:  49 secs]     loss: 0.49933
[epoch:1  iter:8000  elapsed_time:  54 secs]     loss: 0.51417
[epoch:1  iter:8800  elapsed_time:  60 secs]     loss: 0.51097
[epoch:1  iter:9600  elapsed_time:  65 secs]     loss: 0.51736
[epoch:1  iter:10400  elapsed_time:  70 secs]     loss: 0.49400
[epoch:1  iter:11200  elapsed_time:  76 secs]     loss: 0.45961
[epoch:1  iter:12000  elapsed_time:  81 secs]     loss: 0.45388
[epoch:1  iter:12800  elapsed_time:  86 secs]     loss: 0.44298
[epoch:1  iter:13600  elapsed_time:  92 secs]     loss: 0.47452
[epoch:2  iter: 800  elapsed_time: 102 secs]     loss: 0.42375
[epoch:2  iter:1600  elapsed_time: 107 secs]     loss: 0.43689
[epoch:2  iter:2400  elapsed_time: 113 secs]     loss: 0.46617
[epoch:2  iter:3200  elapsed_time: 118 secs]     loss: 0.45911
[epoch:2  iter:4000  elapsed_time: 123 secs]     loss: 0.45125
[epoch:2  iter:4800  elapsed_time: 129 secs]     loss: 0.42654
[epoch:2  iter:5600  elapsed_time: 134 secs]     loss: 0.44630
[epoch:2  iter:6400  elapsed_time: 140 secs]     loss: 0.43953
[epoch:2  iter:7200  elapsed_time: 145 secs]     loss: 0.43179
[epoch:2  iter:8000  elapsed_time: 150 secs]     loss: 0.42519
[epoch:2  iter:8800  elapsed_time: 156 secs]     loss: 0.41138
[epoch:2  iter:9600  elapsed_time: 161 secs]     loss: 0.42337
[epoch:2  iter:10400  elapsed_time: 166 secs]     loss: 0.43318
[epoch:2  iter:11200  elapsed_time: 172 secs]     loss: 0.41804
[epoch:2  iter:12000  elapsed_time: 177 secs]     loss: 0.39491
[epoch:2  iter:12800  elapsed_time: 182 secs]     loss: 0.40882
[epoch:2  iter:13600  elapsed_time: 188 secs]     loss: 0.43010
[epoch:3  iter: 800  elapsed_time: 197 secs]     loss: 0.38005
[epoch:3  iter:1600  elapsed_time: 202 secs]     loss: 0.38328
[epoch:3  iter:2400  elapsed_time: 207 secs]     loss: 0.38381
[epoch:3  iter:3200  elapsed_time: 213 secs]     loss: 0.40059
[epoch:3  iter:4000  elapsed_time: 218 secs]     loss: 0.37907
[epoch:3  iter:4800  elapsed_time: 223 secs]     loss: 0.36653
[epoch:3  iter:5600  elapsed_time: 228 secs]     loss: 0.40294
[epoch:3  iter:6400  elapsed_time: 233 secs]     loss: 0.36507
[epoch:3  iter:7200  elapsed_time: 239 secs]     loss: 0.36799
[epoch:3  iter:8000  elapsed_time: 244 secs]     loss: 0.36584
[epoch:3  iter:8800  elapsed_time: 249 secs]     loss: 0.35412
[epoch:3  iter:9600  elapsed_time: 254 secs]     loss: 0.36120
[epoch:3  iter:10400  elapsed_time: 259 secs]     loss: 0.35645
[epoch:3  iter:11200  elapsed_time: 265 secs]     loss: 0.35363
[epoch:3  iter:12000  elapsed_time: 271 secs]     loss: 0.37596
[epoch:3  iter:12800  elapsed_time: 278 secs]     loss: 0.35256
[epoch:3  iter:13600  elapsed_time: 284 secs]     loss: 0.38427
[epoch:4  iter: 800  elapsed_time: 294 secs]     loss: 0.33827
[epoch:4  iter:1600  elapsed_time: 300 secs]     loss: 0.33363
[epoch:4  iter:2400  elapsed_time: 305 secs]     loss: 0.31443
```

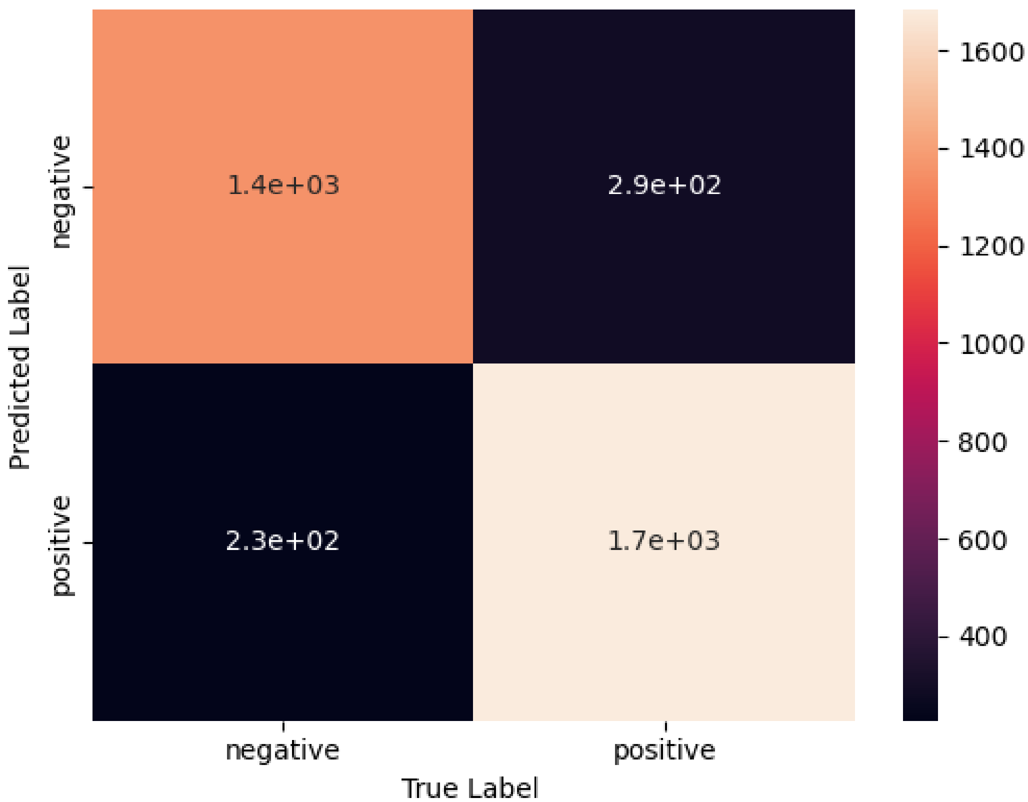
```
[epoch:4  iter:3200  elapsed_time: 310 secs]     loss: 0.33213
[epoch:4  iter:4000  elapsed_time: 315 secs]     loss: 0.36550
[epoch:4  iter:4800  elapsed_time: 320 secs]     loss: 0.34395
[epoch:4  iter:5600  elapsed_time: 324 secs]     loss: 0.33145
[epoch:4  iter:6400  elapsed_time: 329 secs]     loss: 0.34028
[epoch:4  iter:7200  elapsed_time: 334 secs]     loss: 0.29938
[epoch:4  iter:8000  elapsed_time: 339 secs]     loss: 0.36987
[epoch:4  iter:8800  elapsed_time: 344 secs]     loss: 0.32931
[epoch:4  iter:9600  elapsed_time: 349 secs]     loss: 0.34296
[epoch:4  iter:10400  elapsed_time: 355 secs]     loss: 0.32128
[epoch:4  iter:11200  elapsed_time: 360 secs]     loss: 0.32242
[epoch:4  iter:12000  elapsed_time: 365 secs]     loss: 0.30456
[epoch:4  iter:12800  elapsed_time: 371 secs]     loss: 0.34802
[epoch:4  iter:13600  elapsed_time: 377 secs]     loss: 0.31873
[epoch:5  iter: 800  elapsed_time: 386 secs]     loss: 0.28638
[epoch:5  iter:1600  elapsed_time: 391 secs]     loss: 0.30987
[epoch:5  iter:2400  elapsed_time: 396 secs]     loss: 0.29311
[epoch:5  iter:3200  elapsed_time: 401 secs]     loss: 0.29640
[epoch:5  iter:4000  elapsed_time: 407 secs]     loss: 0.31572
[epoch:5  iter:4800  elapsed_time: 412 secs]     loss: 0.31108
[epoch:5  iter:5600  elapsed_time: 418 secs]     loss: 0.29763
[epoch:5  iter:6400  elapsed_time: 423 secs]     loss: 0.31284
[epoch:5  iter:7200  elapsed_time: 428 secs]     loss: 0.27816
[epoch:5  iter:8000  elapsed_time: 434 secs]     loss: 0.29456
[epoch:5  iter:8800  elapsed_time: 440 secs]     loss: 0.30338
[epoch:5  iter:9600  elapsed_time: 445 secs]     loss: 0.33247
[epoch:5  iter:10400  elapsed_time: 450 secs]     loss: 0.31548
[epoch:5  iter:11200  elapsed_time: 455 secs]     loss: 0.28365
[epoch:5  iter:12000  elapsed_time: 460 secs]     loss: 0.31729
[epoch:5  iter:12800  elapsed_time: 466 secs]     loss: 0.28439
[epoch:5  iter:13600  elapsed_time: 472 secs]     loss: 0.30378

No handles with labels found to put in legend.

Lowest loss achieved by network: 0.2782
Training finished in  476 secs
```



Testing

[22]:
```
confusion_matrix, accuracy = test(net2, testloader, log=500, task=2,␣
    ↪bidirectional="bid")
display_confusion_matrix(confusion_matrix, accuracy, task=2,␣
    ↪bidirectional="bid")
```

```
[batch_idx=500]     predicted_label=1       gt_label=1
[batch_idx=1000]     predicted_label=1       gt_label=1
[batch_idx=1500]     predicted_label=0       gt_label=0
[batch_idx=2000]     predicted_label=0       gt_label=0
[batch_idx=2500]     predicted_label=1       gt_label=1
[batch_idx=3000]     predicted_label=1       gt_label=1
[batch_idx=3500]     predicted_label=1       gt_label=1
```



Accuracy: 0.853774908784732

5 Evaluation

The overall accuracies achieved by the custom GRU, the GRU implemented by torch, and the bidirecitonal GRU implemented by torch are

Model	**Accuracy**
GRU scratch	84.1%
Torch GRU	84.2%
Torch Bidirectional GRU	85.3%

Table 2: Hyper-Parameters for Training

From Table 2 we can see that the bidirectional GRU outperformed the ordinary GRU and the custom GRU implemented from scratch. This is because the bidirectional GRU is capable of having the sequence information in both directions: future to past and past to future which does a better job at preserving the future and the past information. The vanilla torch GRU implementation had a better run-time than the birdirectional and custom GRU.