PURDUE UNIVERSITY

Elmore Family School of Electrical and Computer Engineering

Deep Learning

Homework 8

Adithya Sineesh

Email : asineesh@purdue.edu

Submitted: April 17, 2023

1 Experiment

In this homework, we have to create an RNN to classify variable-length Amazon reviews as having positive or negative sentiment using the following GRUs:

1. My own implementation
2. PyTorch's implementation
3. PyTorch's bidirectional implementation

I have referred to Prof. Kak's implementations on DLStudio.py for this assignment.

1.1 Dataset

I first downloaded the training (sentiment_dataset_train_400.tar.gz) and validation (sentiment_dataset_test_400.tar.gz) dataset from the test datasets for DLStudio. It contains 14227 training and 3563 validation reviews across 25 categories along with the corresponding sentiment label (positive or negative).

For the word embeddings, I used the pre-trained word2vec which consists of 300 element vectors for 3 million words and phrases.

1.2 Model Training

The architecture of all the 3 RNNs includes their respective GRUs followed by ReLU activation. Then it is passed through a Linear layer (number of output nodes is 2) and logsoftmax activation.

The optimizer used was Adam with beta values of (0.9,0.99), the learning rate was 0.001 for 6 epochs and the batch size was 1. The loss function was Negative Log Likelihood Loss.

For all the GRUs, the input size was 300; hidden size was 100. For my implementation of the GRU, the number of layers was 1 and it was 2 for the other 2 GRUs. The hidden state at the start of a review was always initialized to 0.

The logic for all the three GRUs is as follows:

1. **My own implementation**

   It is influenced by Prof Kak's implementation of pmGRU. The following are the equations involved in it:

$$f = \sigma(W_f x_t + U_f h_{t-1})$$
$$\tilde{h_t} = \sigma(W_h x_t + U_h(f \cdot h_{t-1})$$
$$h_t = (1 - f) \cdot h_{t-1} + f \cdot \tilde{h_t}$$

   where,

   f is the forget gate

   x_t is the current input

   h_{t-1} is the previous hidden state $\tilde{h_t}$ is the candidate hidden state

   h_t is the final hidden state/output at t

   $\sigma()$ is the activation function

   $\cdot$ is the elementwise product

   W_f, U_f, W_h, U_h are the learnable linear layers

   Usually in a RNN, there exists a long chain of dependencies spanning across each hidden state at all the steps. As this leads to a very deep neural network, the threat of vanishing gradients is exacerbated. In GRUs there exist gates, which learn to forget or update information at each step based on the current input and the previous hidden state. This essentially provides multiple paths for the gradient to backpropagate, thereby reducing the problem of vanishing gradients.

2. **PyTorch's unidirectional implementation**

   For this, we use the torch.nn.GRU() module. Its instance is defined by specifying the input size, hidden size and the number of layers. The inputs of its instance are of shape:

   (a) input $\longrightarrow$ (seq_len, batch_size, in_size)

   (b) hidden $\longrightarrow$ (num_layers, batch_size, hidden_size)

   If seq_len is 1, then we feed each review, word by word, to the GRU (which is what I have done for this assignment). Else if the seq_len is the length of the review, we feed the entire review to the GRU at once (which is much faster).

   The outputs of a torch.nn.GRU() instance are of shape:

   (a) output $\longrightarrow$ (seq_len, batch_size, hidden_size)

   (b) hidden $\longrightarrow$ (num_layers, batch_size, hidden_size)

3. **PyTorch's biidirectional implementation**

   In this case the instance of torch.nn.GRU() module is defined by setting an extra parameter *bidirectional* to True. The entire review is also fed at once into the GRU. The inputs of its instance are of shape:

   (a) input $\longrightarrow$ (seq_len, batch_size, in_size)

   (b) hidden $\longrightarrow$ (2*num_layers, batch_size, hidden_size)

   The outputs of the instance are of shape:

   (a) output $\longrightarrow$ (seq_len, batch_size, 2*hidden_size)

   (b) hidden $\longrightarrow$ (2*num_layers, batch_size, hidden_size)

1.3 Evaluation

This is done quantitatively by:

1. Plotting the confusion matrix along with the accuracy.

2. Plotting the loss vs epochs graph.

3. Plotting the test accuracy vs epochs graph

2 Results

2.1 For my implementation of GRU

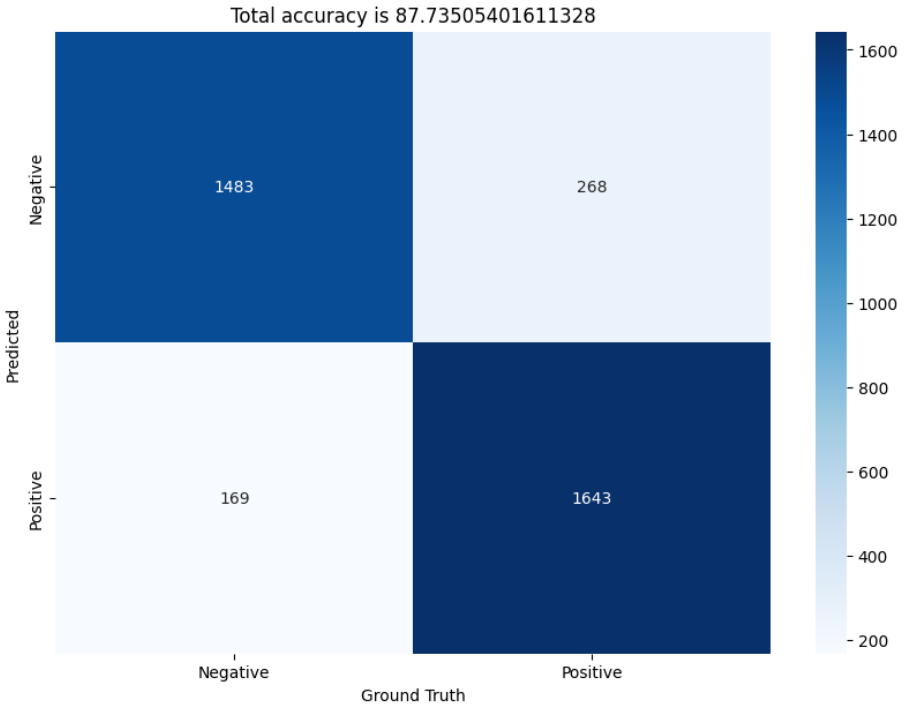
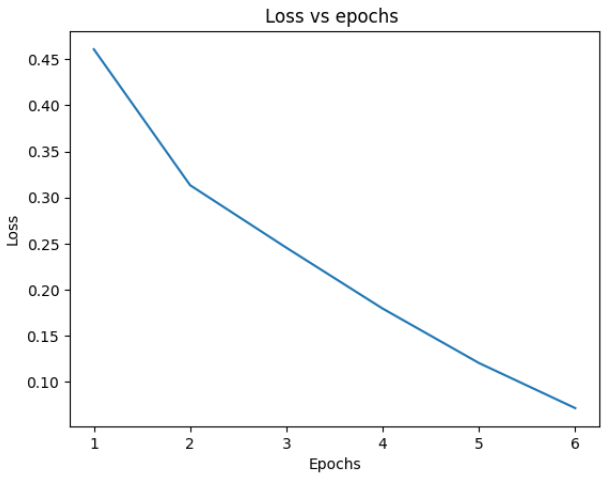
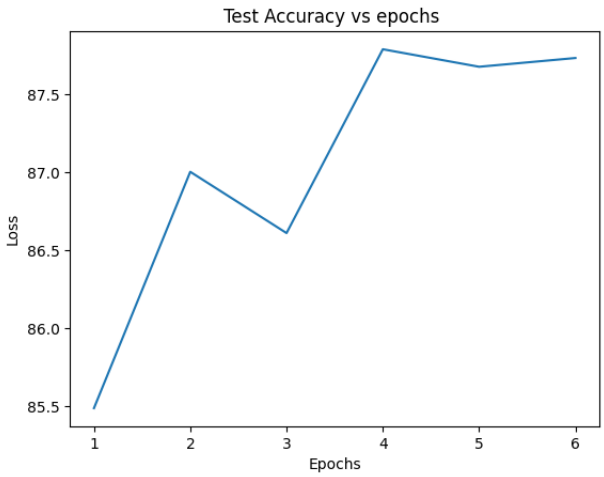


Figure 1: Confusion Matrix





2.2 For PyTorch's implementation of unidirectional GRU

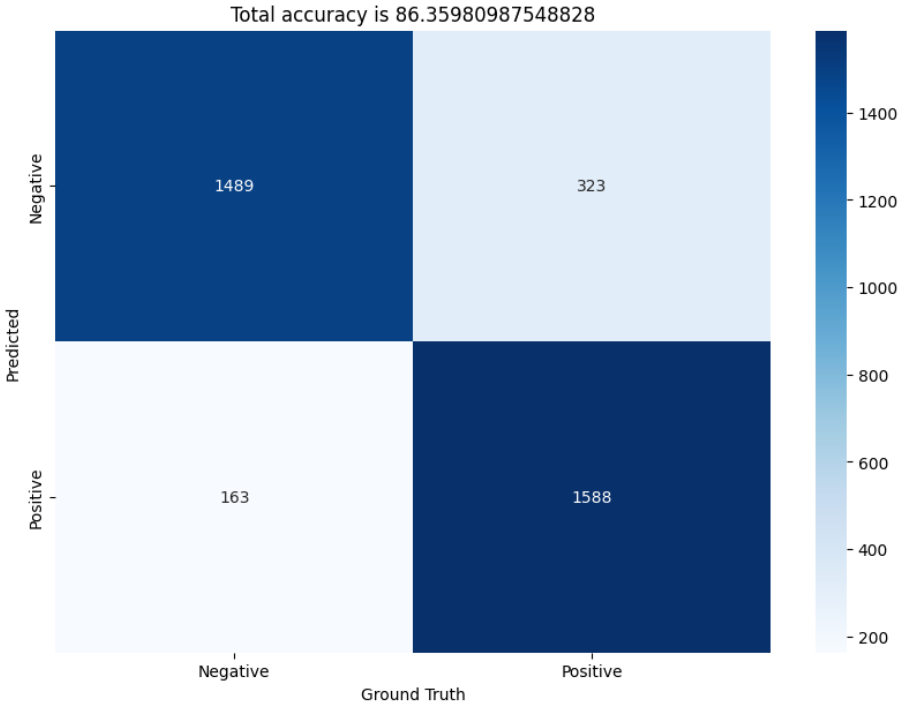
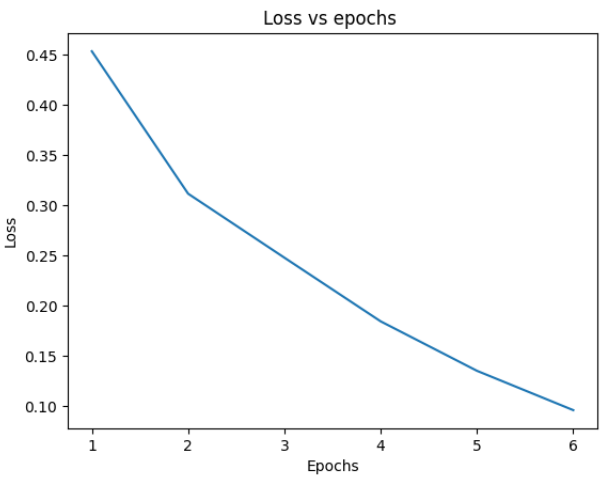
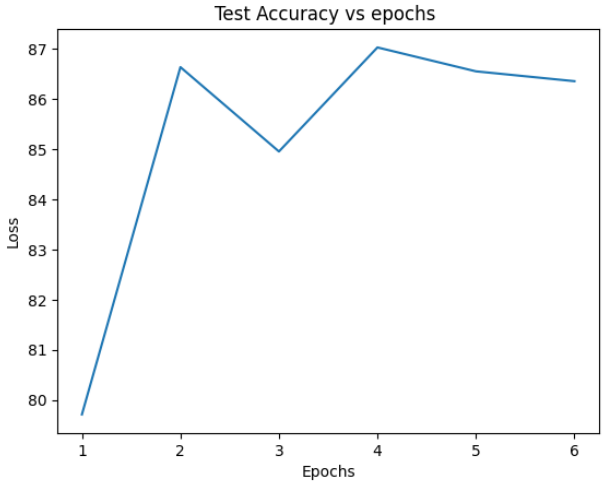


Figure 2: Confusion Matrix





2.3 For PyTorch's implementation of bidirectional GRU

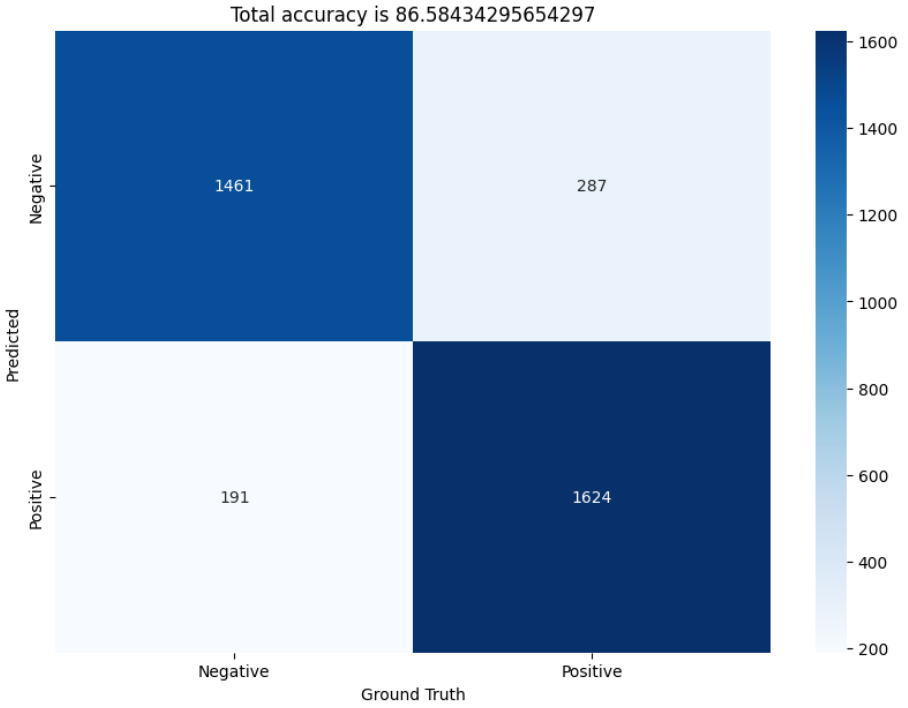
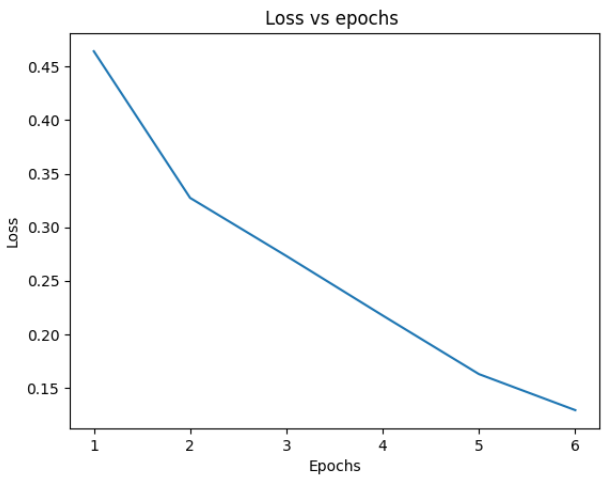
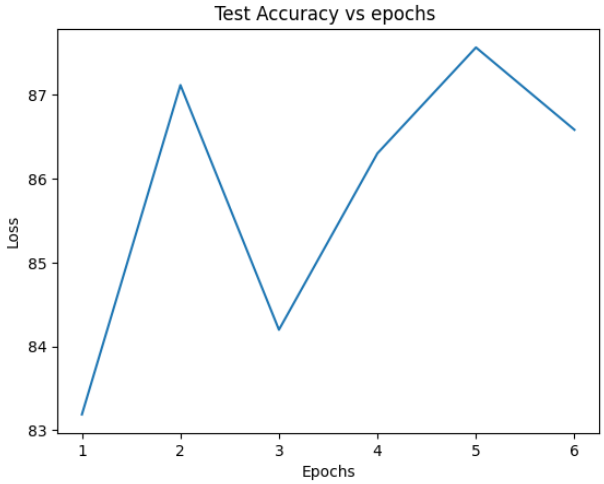


Figure 3: Confusion Matrix





3 Conclusion

Quantitatively from best to worst, the test accuracy for my implementation of GRU is 87.735%, PyTorch's implementation of bidirectional GRU is 86.584% and PyTorch's implementation of unidirectional GRU is 86.360%. But by looking at the Test Accuracy vs Epochs graph for all three models, it is clear that overfitting occurs as the dataset is quite small.

In this assignment, the bidirectional GRU does give better test result compared to the unidirectional GRU. But one can't read too much into this due to the reason mentioned above.

In [3]:

```
#Dowloading the embeddings
import gensim.downloader as gen_api
from gensim.models import KeyedVectors
```

In [5]:

```
word_vectors = gen_api.load("word2vec-google-news-300")
word_vectors.save( 'vectors.kv')
```

In [ ]:

```

```

In [1]:

```
#Importing the libraries
import torch
import torch.nn as nn
import os
import sys
import pickle
import random
import numpy as np
import gzip
import gensim.downloader as gen_api
from gensim.models import KeyedVectors
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from tqdm import tqdm
```

In [2]:

```
#Dataset definition from DL Studio
class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, dataset_file, path_to_saved_embeddings):
        super(SentimentAnalysisDataset, self).__init__()

        f = gzip.open(dataset_file, 'rb') #Opening the compressed dataset file
        dataset = f.read() #dataset contains the binary objects of the file
        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv') #

        #Converting the binary string to Python objects
        self.positive_reviews, self.negative_reviews, self.vocab = pickle.loads(dataset,
        #1. Vocab is a list of all the unique words in the dataset
        #2. positive_reviews and negative_reviews is a hashmap consisting of 25 keys cor
        # different categories. The value corresponding to the category key consists of
        # each row corresponding to a specific review and the column of that row corresp

        #Storing each review along with its category and sentiment in a list
        self.indexed_dataset = []
        for category in self.positive_reviews:             #Iterating through all the co
            for review in self.positive_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 1]) #1 indicates positive
        for category in self.negative_reviews:             #Iterating through all the co
            for review in self.negative_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 0]) #0 indicates negative

    def review_to_tensor(self, review):
        #Converts each word of the review into its embedding of shape 300
        list_of_embeddings = []
        for word in review: #Iterating through each word in the review
            if word in self.word_vectors.key_to_index: #Checking if the embedding for th
                embedding = self.word_vectors[word]
                list_of_embeddings.append(np.array(embedding)) #Appending its embedding
            else:
                next
        #Converting the list to a tensor
        review_array = np.array(list_of_embeddings)
        review_tensor = torch.from_numpy(review_array)
        return review_tensor #Its of shape (num_words x 300)

    def __len__(self):
        #Returns the length of the dataset
        return len(self.indexed_dataset)

    def __getitem__(self, idx):
        #Extracts an individual review along with its category and sentiment
        sample = self.indexed_dataset[idx]
        review = sample[0]
        review_category = sample[1]
        review_sentiment = sample[2]

        #Converts the review words to its embeddings for each word of the review
        review_tensor = self.review_to_tensor(review)

        #Wrapping up the review, category and sentiment in a hashmap
        sample = {'review'       : review_tensor,
                  'sentiment'    : review_sentiment }
        return sample
```

In [3]:

```
class mGRU(nn.Module):
    def __init__(self, input_size, hidden_size,output_size,batch_size):
        super().__init__()
        self.Linear1 = nn.Linear(input_size + hidden_size, hidden_size)
        self.sigmoid = nn.Sigmoid()
        self.Linear2 = nn.Linear(input_size + hidden_size, hidden_size)
        self.tanh =  nn.Tanh()

    def forward(self, x, h):
        combined1 = torch.cat((x, h), 2)
        forget = self.sigmoid(self.Linear1(combined1))
        interim_h =  forget*h
        combined2  = torch.cat((x, interim_h), 2)
        interim_o =  self.tanh(self.Linear2(combined2))
        output = (1 - forget)*h  +  forget*interim_o
        return output, output
```

In [4]:

```
#Model architecture from DLStudio
class GRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru =  mGRU(input_size,hidden_size,output_size,1) #Using pmGRU
        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_new = self.gru(x, h)
        fout = self.fc(self.relu(out[:,-1]))
        fout = self.logsoftmax(fout)
        return fout, h_new

    def init_hidden(self):
        #Creates the first hidden input which is all zeros
        weight = next(self.parameters()).data
        hidden = weight.new(self.num_layers,1,self.hidden_size).zero_()
        return hidden
```

In [5]:

```
def testing(net,device,test_dataloader):
  test_corr = 0
  net.eval()
  with torch.no_grad():
    loop = tqdm(test_dataloader)
    for i,data in enumerate(loop):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = net.init_hidden().to(device)

       #Iterating through all the words in the sentence
       for k in range(review_tensor.shape[1]):
          #Passing word by word through the model
          output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0),

       predicted = torch.max(output.data, 1)[1]
       test_corr += (predicted == sentiment).item()

  return (test_corr*100)/len(test_dataloader.dataset)
```

In [6]:

```
#From DLStudio
def training(net, criterion, optimizer, epochs, train_dataloader, test_dataloader, devic
    training_loss = [] #Keeping track of the loss across epochs
    test_accs = [] #Keeping track of test accuracies across epochs

    #Iterating through the epochs
    for epoch in range(epochs):
        running_loss = 0.0
        loop = tqdm(train_dataloader)
        #Iterating through the dataloader
        for i, data in enumerate(loop):
            #Extracting the review and sentiment
            review_tensor,sentiment = data['review'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            optimizer.zero_grad()

            #Initializing the hidden to 0
            hidden = net.init_hidden().to(device)

            #Iterating through all the words in the sentence
            for k in range(review_tensor.shape[1]):
                #Passing word by word through the model
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],
                #input is of shape (1,1,300)
                #hidden is of shape (1,1,100)
            #Computing loss
            loss = criterion(output,sentiment)
            running_loss += loss.item()
            loop.set_postfix(loss=loss.item())
            loss.backward()
            optimizer.step()

        training_loss.append((running_loss)/(i+1))
        test_acc = testing(net,device,test_dataloader)
        test_accs.append(test_acc)
        net.train()
        print("[epoch: %d, batch: %5d] Loss: %.3f Test Accuracy: %.3f " % (epoch + 1, i


    return net, training_loss, test_accs
```

In [7]:

```
#Function to compute the confusion matrix
def confusion_matrix(model,test_data_loader,device):
  matrix = torch.zeros((2,2))
  model.eval()
  with torch.no_grad():
    for i,data in enumerate(test_dataloader):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = model.init_hidden().to(device)

       #Iterating through all the words in the sentence
       for k in range(review_tensor.shape[1]):
          #Passing word by word through the model
          output, hidden = model(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0

       predicted = torch.max(output.data, 1)[1]
       matrix[predicted.cpu(),sentiment.cpu()] += 1

  heat = pd.DataFrame(matrix, index = [i for i in ["Negative","Positive"]],
                      columns = [i for i in ["Negative","Positive"]])
  heat = heat.astype(int)
  accuracy = (matrix.trace()/matrix.sum())*100
  plt.figure(figsize = (10,7))
  plt.title("Total accuracy is "+str(accuracy.item()))
  s = sn.heatmap(heat, annot=True,cmap='Blues',fmt='g')
  s.set(xlabel='Ground Truth', ylabel='Predicted')
```

In [8]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[8]:

```
device(type='cuda')
```

In [9]:

```
model = GRUnet(input_size=300,hidden_size=100,output_size=2,num_layers=1).to(device)
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,betas=(0.9,0.99))
epochs = 6
```

In [10]:

```
train_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/se
train_dataloader = torch.utils.data.DataLoader(train_dataset,batch_size=1,shuffle=True,

test_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/ser
test_dataloader = torch.utils.data.DataLoader(test_dataset,batch_size=1,shuffle=True, nu

print("The size of the training dataset is "+ str(len(train_dataset)))
print("The size of the test dataset is "+ str(len(test_dataset)))
```

```
The size of the training dataset is 14227
The size of the test dataset is 3563
```

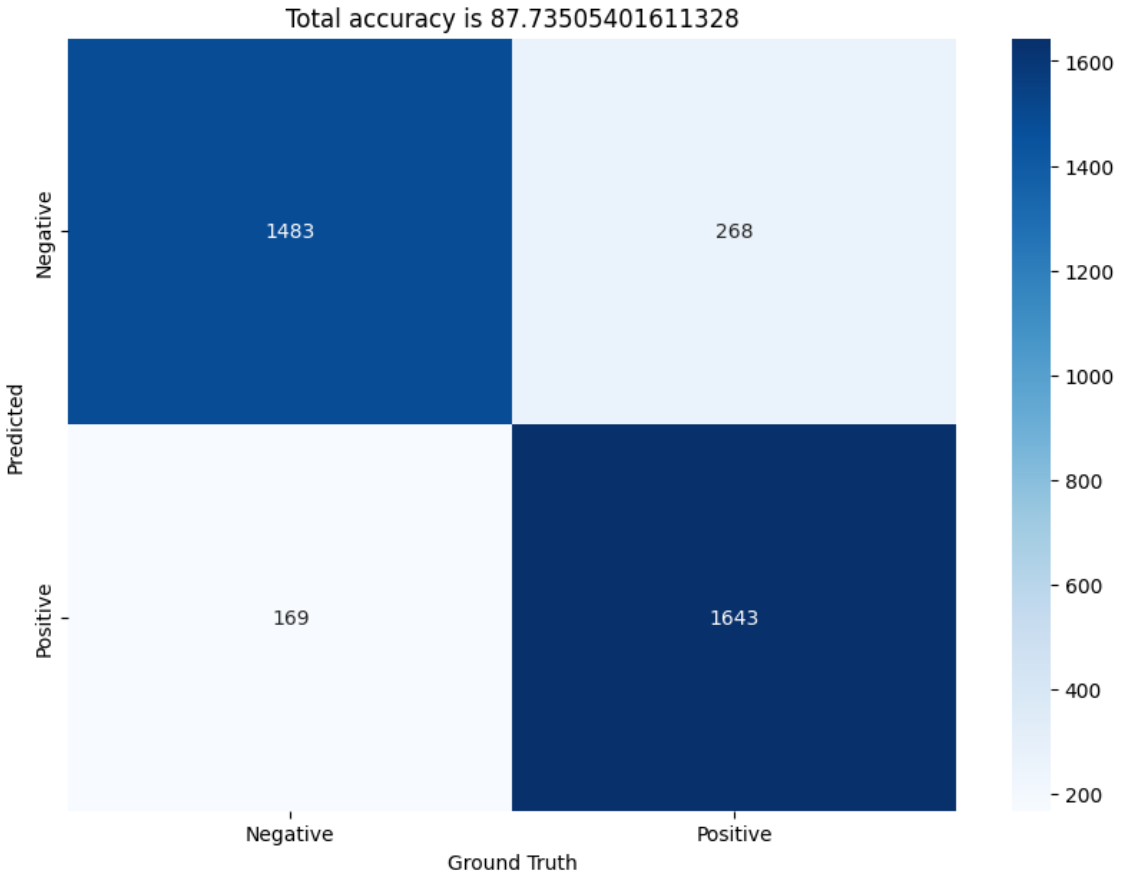
In [11]:

```
trained_model, training_loss, test_accs = training(model, criterion, optimizer, epochs,
```

```
100%|██████████| 14227/14227 [18:52<00:00, 12.56it/s, loss=0.0795]
100%|██████████| 3563/3563 [02:00<00:00, 29.49it/s]

[epoch: 1, batch: 14227] Loss: 0.461 Test Accuracy: 85.490

100%|██████████| 14227/14227 [18:47<00:00, 12.62it/s, loss=7.73]
100%|██████████| 3563/3563 [01:59<00:00, 29.83it/s]

[epoch: 2, batch: 14227] Loss: 0.313 Test Accuracy: 87.005

100%|██████████| 14227/14227 [18:44<00:00, 12.65it/s, loss=2.14]
100%|██████████| 3563/3563 [01:59<00:00, 29.82it/s]

[epoch: 3, batch: 14227] Loss: 0.246 Test Accuracy: 86.612

100%|██████████| 14227/14227 [18:21<00:00, 12.91it/s, loss=0.00265]
100%|██████████| 3563/3563 [01:58<00:00, 30.11it/s]

[epoch: 4, batch: 14227] Loss: 0.180 Test Accuracy: 87.791

100%|██████████| 14227/14227 [19:15<00:00, 12.31it/s, loss=0.0183]
100%|██████████| 3563/3563 [02:05<00:00, 28.45it/s]

[epoch: 5, batch: 14227] Loss: 0.121 Test Accuracy: 87.679

100%|██████████| 14227/14227 [19:20<00:00, 12.26it/s, loss=0.1]
100%|██████████| 3563/3563 [02:02<00:00, 29.07it/s]

[epoch: 6, batch: 14227] Loss: 0.072 Test Accuracy: 87.735
```

In [12]:

```
confusion_matrix(trained_model,test_dataloader,device)
```

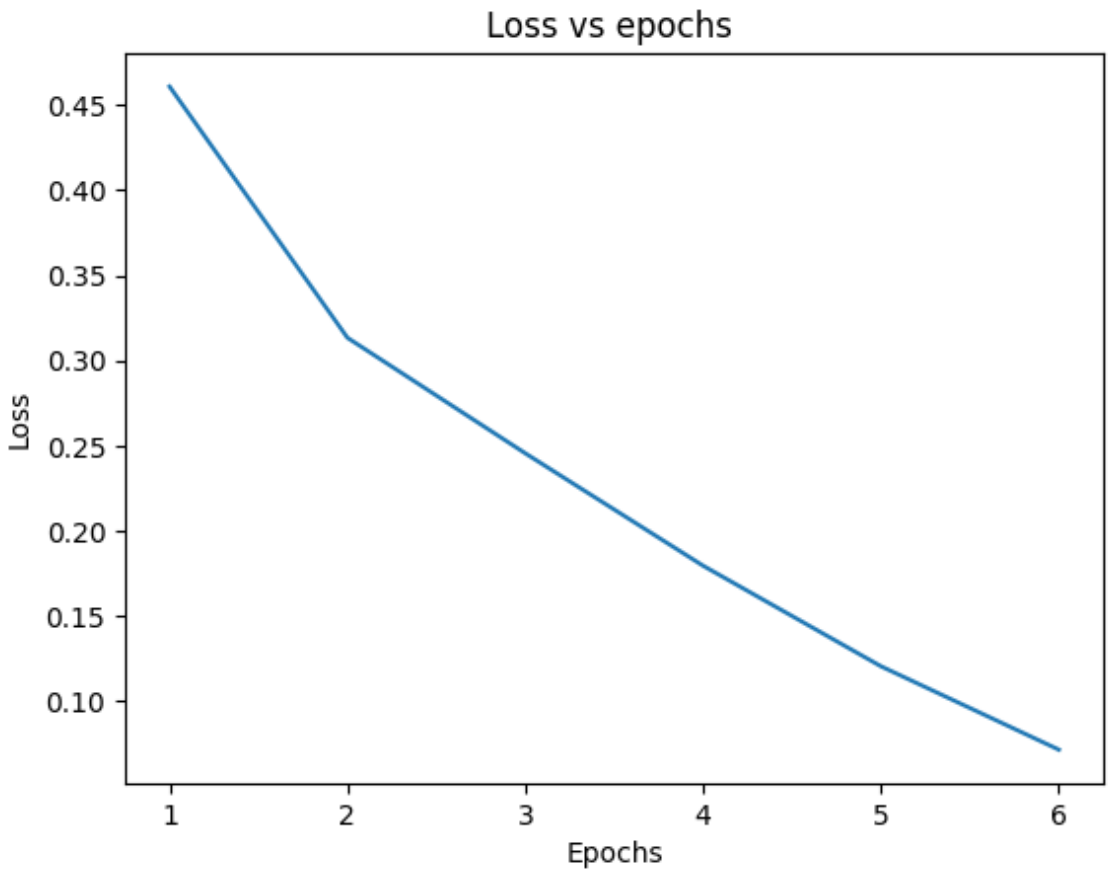


In [13]:

```
#Plotting the training loss vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs epochs ")
plt.plot(epochs,training_loss)
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x7fed69b63d60>]
```

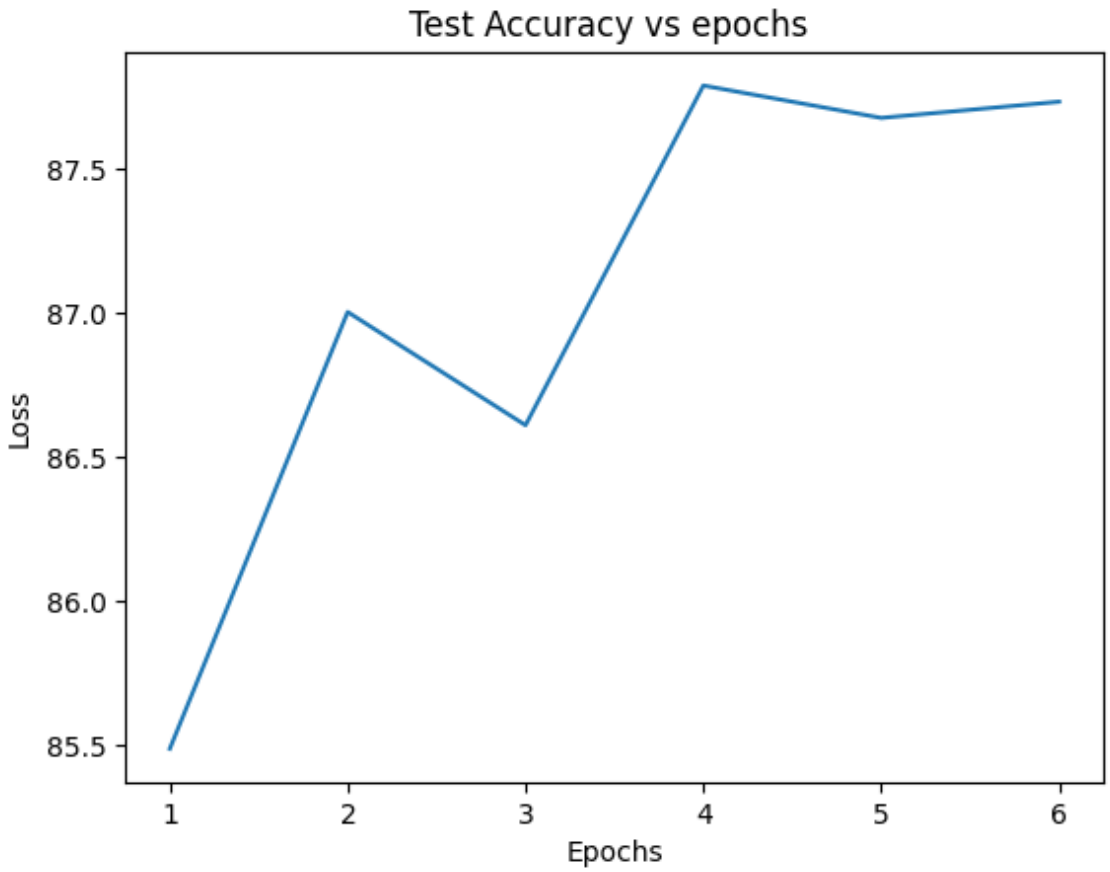


In [14]:

```
#Plotting the test accuracy vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Test Accuracy vs epochs ")
plt.plot(epochs,test_accs)
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x7fed69a9b820>]
```



In [15]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

In [16]:

```
#Number of parameters in the model
count_parameters(trained_model)
```

Out[16]:

```
80402
```

In [ ]:

```
#Importing the libraries
import torch
import torch.nn as nn
import os
import sys
import pickle
import random
import numpy as np
import gzip
import gensim.downloader as gen_api
from gensim.models import KeyedVectors
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from tqdm import tqdm
```

In [ ]:

```
#Dataset definition from DL Studio
class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, dataset_file, path_to_saved_embeddings):
        super(SentimentAnalysisDataset, self).__init__()

        f = gzip.open(dataset_file, 'rb') #Opening the compressed dataset file
        dataset = f.read() #dataset contains the binary objects of the file
        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv') #

        #Converting the binary string to Python objects
        self.positive_reviews, self.negative_reviews, self.vocab = pickle.loads(dataset,
        #1. Vocab is a list of all the unique words in the dataset
        #2. positive_reviews and negative_reviews is a hashmap consisting of 25 keys cor
        # different categories. The value corresponding to the category key consists of
        # each row corresponding to a specific review and the column of that row corresp

        #Storing each review along with its category and sentiment in a list
        self.indexed_dataset = []
        for category in self.positive_reviews:             #Iterating through all the co
            for review in self.positive_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 1]) #1 indicates positive
        for category in self.negative_reviews:             #Iterating through all the co
            for review in self.negative_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 0]) #0 indicates negative

    def review_to_tensor(self, review):
        #Converts each word of the review into its embedding of shape 300
        list_of_embeddings = []
        for word in review: #Iterating through each word in the review
            if word in self.word_vectors.key_to_index: #Checking if the embedding for th
                embedding = self.word_vectors[word]
                list_of_embeddings.append(np.array(embedding)) #Appending its embedding
            else:
                next
        #Converting the list to a tensor
        review_array = np.array(list_of_embeddings)
        review_tensor = torch.from_numpy(review_array)
        return review_tensor #Its of shape (num_words x 300)

    def __len__(self):
        #Returns the length of the dataset
        return len(self.indexed_dataset)

    def __getitem__(self, idx):
        #Extracts an individual review along with its category and sentiment
        sample = self.indexed_dataset[idx]
        review = sample[0]
        review_category = sample[1]
        review_sentiment = sample[2]

        #Converts the review words to its embeddings for each word of the review
        review_tensor = self.review_to_tensor(review)

        #Wrapping up the review, category and sentiment in a hashmap
        sample = {'review'       : review_tensor,
                  'sentiment'    : review_sentiment }
        return sample
```

In [ ]:

```
#Model architecture from DL Studio
class GRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers) #Using PyTorch's implemer
        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):
        #Creates the first hidden input which is all zeros
        weight = next(self.parameters()).data
        #                                       batch_size
        hidden = weight.new(  self.num_layers,     1,         self.hidden_size    ).zero_
        return hidden
```

In [ ]:

```
def testing(net,device,test_dataloader):
  test_corr = 0
  net.eval()
  with torch.no_grad():
    loop = tqdm(test_dataloader)
    for i,data in enumerate(loop):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = net.init_hidden().to(device)

       #Iterating through all the words in the sentence
       for k in range(review_tensor.shape[1]):
          #Passing word by word through the model
          output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0),

       predicted = torch.max(output.data, 1)[1]
       test_corr += (predicted == sentiment).item()

  return (test_corr*100)/len(test_dataloader.dataset)
```

In [ ]:

```
#From DL studio
def training(net, criterion, optimizer, epochs, train_dataloader, test_dataloader, devic
    training_loss = [] #Keeping track of the loss across epochs
    test_accs = [] #Keeping track of test accuracies across epochs

    #Iterating through the epochs
    for epoch in range(epochs):
        running_loss = 0.0
        loop = tqdm(train_dataloader)
        #Iterating through the dataloader
        for i, data in enumerate(loop):
            #Extracting the review and sentiment
            review_tensor,sentiment = data['review'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            optimizer.zero_grad()

            #Initializing the hidden to 0
            hidden = net.init_hidden().to(device)

            #Iterating through all the words in the sentence
            for k in range(review_tensor.shape[1]):
                #Passing word by word through the model
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],
                #input is of shape (1,1,300)
                #hidden is of shape (1,1,100)
            #Computing loss
            loss = criterion(output,sentiment)
            running_loss += loss.item()
            loop.set_postfix(loss=loss.item())
            loss.backward()
            optimizer.step()

        training_loss.append((running_loss)/(i+1))
        test_acc = testing(net,device,test_dataloader)
        test_accs.append(test_acc)
        net.train()
        print("[epoch: %d, batch: %5d] Loss: %.3f Test Accuracy: %.3f " % (epoch + 1, i


    return net, training_loss, test_accs
```

In [ ]:

```
#Function to compute the confusion matrix
def confusion_matrix(model,test_data_loader,device):
  matrix = torch.zeros((2,2))
  model.eval()
  with torch.no_grad():
    for i,data in enumerate(test_dataloader):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = model.init_hidden().to(device)

       #Iterating through all the words in the sentence
       for k in range(review_tensor.shape[1]):
          #Passing word by word through the model
          output, hidden = model(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0

       predicted = torch.max(output.data, 1)[1]
       matrix[predicted.cpu(),sentiment.cpu()] += 1

  heat = pd.DataFrame(matrix, index = [i for i in ["Negative","Positive"]],
                  columns = [i for i in ["Negative","Positive"]])
  heat = heat.astype(int)
  accuracy = (matrix.trace()/matrix.sum())*100
  plt.figure(figsize = (10,7))
  plt.title("Total accuracy is "+str(accuracy.item()))
  s = sn.heatmap(heat, annot=True,cmap='Blues',fmt='g')
  s.set(xlabel='Ground Truth', ylabel='Predicted')
```

In [ ]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[7]:

```
device(type='cuda')
```

In [ ]:

```
model = GRUnet(input_size=300,hidden_size=100,output_size=2,num_layers=2).to(device)
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,betas=(0.9,0.99))
epochs = 6
```

In [ ]:

```
train_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/se
train_dataloader = torch.utils.data.DataLoader(train_dataset,batch_size=1,shuffle=True,

test_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/ser
test_dataloader = torch.utils.data.DataLoader(test_dataset,batch_size=1,shuffle=True, nu

print("The size of the training dataset is "+ str(len(train_dataset)))
print("The size of the test dataset is "+ str(len(test_dataset)))
```

```
The size of the training dataset is 14227
The size of the test dataset is 3563
```

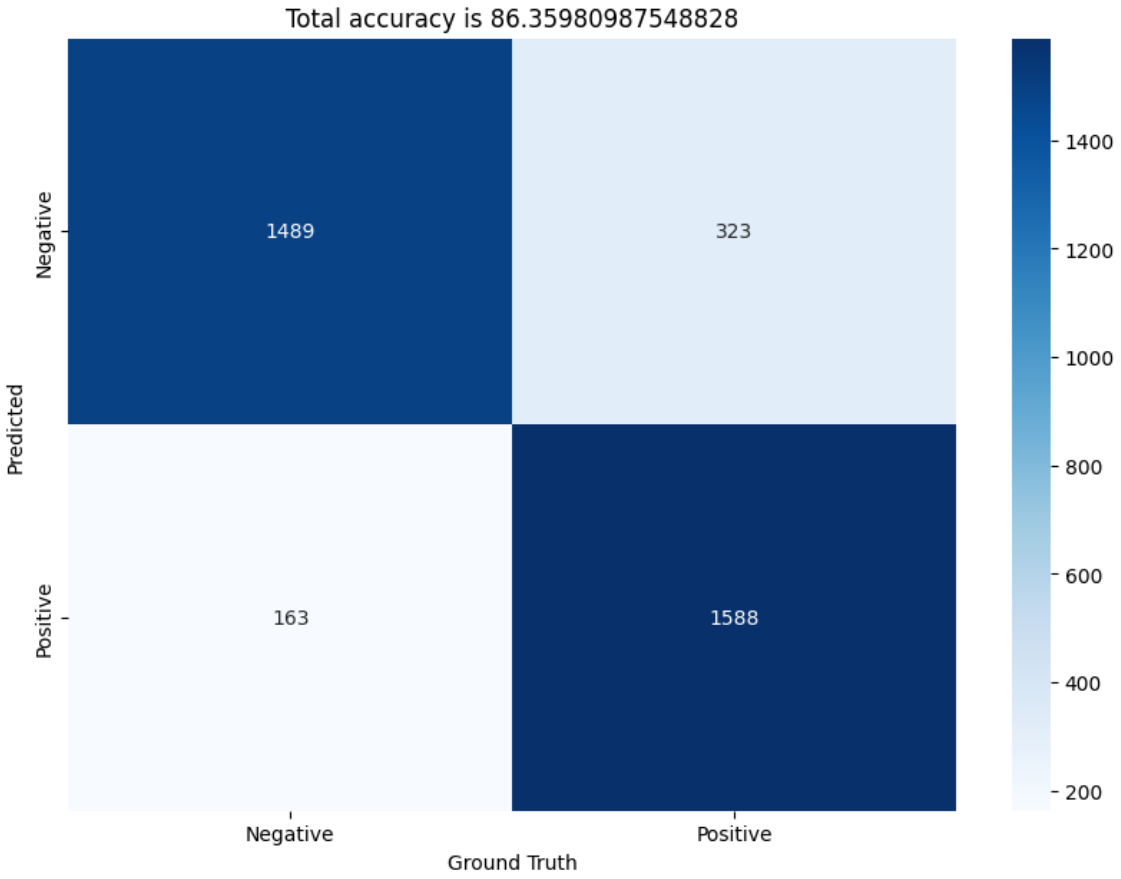
In [ ]:

```
trained_model, training_loss, test_accs = training(model, criterion, optimizer, epochs,
```

```
100%|██████████| 14227/14227 [15:47<00:00, 15.01it/s, loss=0.417]
100%|██████████| 3563/3563 [01:47<00:00, 33.18it/s]

[epoch: 1, batch: 14227] Loss: 0.453 Test Accuracy: 79.708

100%|██████████| 14227/14227 [15:53<00:00, 14.92it/s, loss=0.0274]
100%|██████████| 3563/3563 [01:48<00:00, 32.94it/s]

[epoch: 2, batch: 14227] Loss: 0.312 Test Accuracy: 86.640

100%|██████████| 14227/14227 [15:49<00:00, 14.98it/s, loss=0.0156]
100%|██████████| 3563/3563 [01:48<00:00, 32.98it/s]

[epoch: 3, batch: 14227] Loss: 0.248 Test Accuracy: 84.956

100%|██████████| 14227/14227 [15:51<00:00, 14.95it/s, loss=0.012]
100%|██████████| 3563/3563 [01:47<00:00, 33.22it/s]

[epoch: 4, batch: 14227] Loss: 0.184 Test Accuracy: 87.033

100%|██████████| 14227/14227 [15:53<00:00, 14.92it/s, loss=0.000211]
100%|██████████| 3563/3563 [01:48<00:00, 32.80it/s]

[epoch: 5, batch: 14227] Loss: 0.135 Test Accuracy: 86.556

100%|██████████| 14227/14227 [16:00<00:00, 14.82it/s, loss=0.000796]
100%|██████████| 3563/3563 [01:50<00:00, 32.28it/s]

[epoch: 6, batch: 14227] Loss: 0.096 Test Accuracy: 86.360
```

In [ ]:

```
confusion_matrix(trained_model,test_dataloader,device)
```

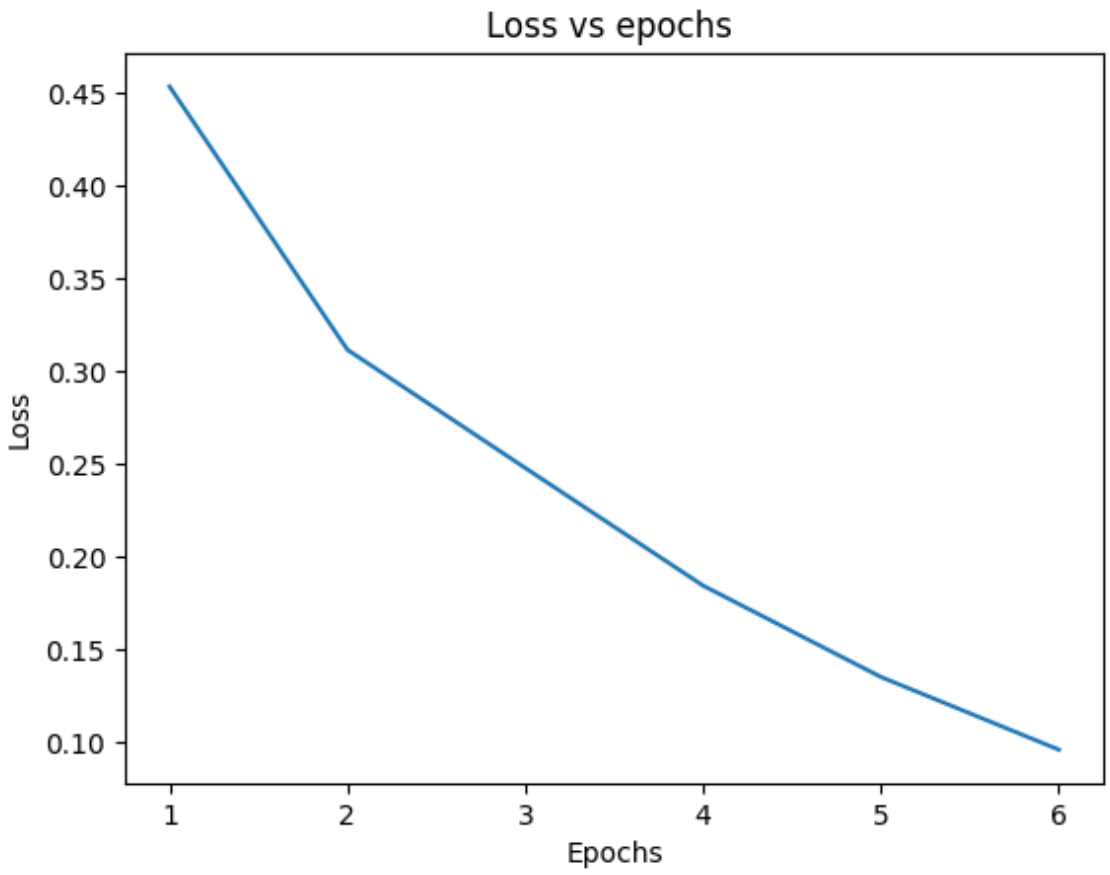


In [ ]:

```
#Plotting the training loss vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs epochs ")
plt.plot(epochs,training_loss)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x7f86019beb20>]
```

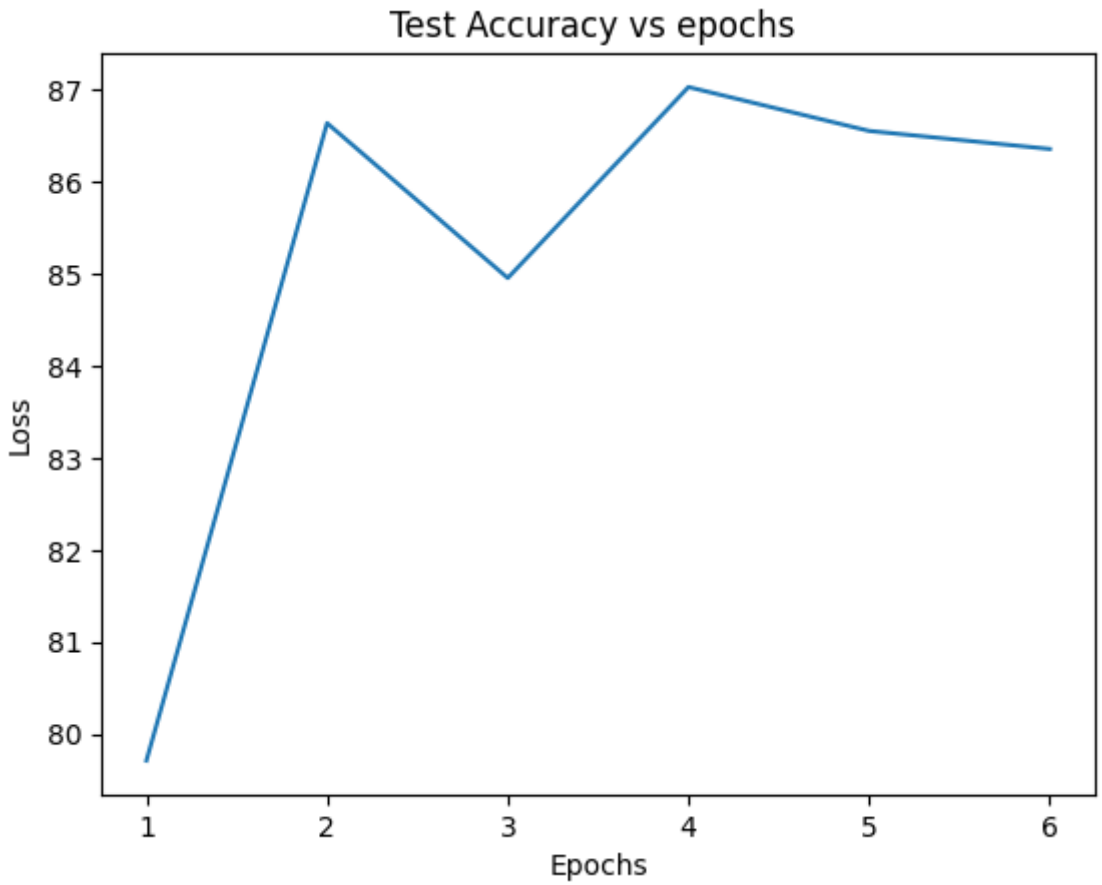


In [ ]:

```
#Plotting the test accuracy vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Test Accuracy vs epochs ")
plt.plot(epochs,test_accs)
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x7f860183cfd0>]
```



In [ ]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

In [ ]:

```
#Number of parameters
count_parameters(trained_model)
```

Out[15]:

```
181402
```

In [ ]:

```
```

In [1]:

```
#Importing the libraries
import torch
import torch.nn as nn
import os
import sys
import pickle
import random
import numpy as np
import gzip
import gensim.downloader as gen_api
from gensim.models import KeyedVectors
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from tqdm import tqdm
```

In [2]:

```
#Dataset definition from DLStudio
class SentimentAnalysisDataset(torch.utils.data.Dataset):
    def __init__(self, dataset_file, path_to_saved_embeddings):
        super(SentimentAnalysisDataset, self).__init__()

        f = gzip.open(dataset_file, 'rb') #Opening the compressed dataset file
        dataset = f.read() #dataset contains the binary objects of the file
        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv') #

        #Converting the binary string to Python objects
        self.positive_reviews, self.negative_reviews, self.vocab = pickle.loads(dataset,
        #1. Vocab is a list of all the unique words in the dataset
        #2. positive_reviews and negative_reviews is a hashmap consisting of 25 keys cor
        # different categories. The value corresponding to the category key consists of
        # each row corresponding to a specific review and the column of that row corresp

        #Storing each review along with its category and sentiment in a list
        self.indexed_dataset = []
        for category in self.positive_reviews:             #Iterating through all the co
            for review in self.positive_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 1]) #1 indicates positive
        for category in self.negative_reviews:             #Iterating through all the co
            for review in self.negative_reviews[category]: #Iterating through all the re
                self.indexed_dataset.append([review, category, 0]) #0 indicates negative

    def review_to_tensor(self, review):
        #Converts each word of the review into its embedding of shape 300
        list_of_embeddings = []
        for word in review: #Iterating through each word in the review
            if word in self.word_vectors.key_to_index: #Checking if the embedding for th
                embedding = self.word_vectors[word]
                list_of_embeddings.append(np.array(embedding)) #Appending its embedding
            else:
                next
        #Converting the list to a tensor
        review_array = np.array(list_of_embeddings)
        review_tensor = torch.from_numpy(review_array)
        return review_tensor #Its of shape (num_words x 300)

    def __len__(self):
        #Returns the length of the dataset
        return len(self.indexed_dataset)

    def __getitem__(self, idx):
        #Extracts an individual review along with its category and sentiment
        sample = self.indexed_dataset[idx]
        review = sample[0]
        review_category = sample[1]
        review_sentiment = sample[2]

        #Converts the review words to its embeddings for each word of the review
        review_tensor = self.review_to_tensor(review)

        #Wrapping up the review, category and sentiment in a hashmap
        sample = {'review'       : review_tensor,
                  'sentiment'    : review_sentiment }
        return sample
```

In [3]:

```
#Model architecture from DLStudio
class GRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers,bidirectional = True) #Usi
        self.fc = nn.Linear(2*hidden_size, output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        out, h_new = self.gru(x, h)
        final_o = torch.cat((h_new[2,:,:],h_new[3,:,:]),1) #Concatenating the output of
        fout = self.fc(self.relu(final_o))
        fout = self.logsoftmax(fout)
        return fout, h_new

    def init_hidden(self):
        #Creates the first hidden input which is all zeros
        weight = next(self.parameters()).data
        hidden = weight.new( 2*self.num_layers, 1,self.hidden_size).zero_()
        return hidden
```

In [4]:

```
def testing(net,device,test_dataloader):
  test_corr = 0
  net.eval()
  with torch.no_grad():
    loop = tqdm(test_dataloader)
    for i,data in enumerate(loop):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = net.init_hidden().to(device)

       #Passing the entire review through the model
       output, hidden = net(torch.unsqueeze(review_tensor[0],1), hidden)
       predicted = torch.max(output.data, 1)[1]
       test_corr += (predicted == sentiment).item()

  return (test_corr*100)/len(test_dataloader.dataset)
```

In [5]:

```
#From DLStudio
def training(net, criterion, optimizer, epochs, train_dataloader, test_dataloader, devic
    training_loss = [] #Keeping track of the loss across epochs
    test_accs = [] #Keeping track of test accuracies across epochs

    #Iterating through the epochs
    for epoch in range(epochs):
        running_loss = 0.0
        loop = tqdm(train_dataloader)
        #Iterating through the dataloader
        for i, data in enumerate(loop):
            #Extracting the review and sentiment
            review_tensor,sentiment = data['review'], data['sentiment']
            review_tensor = review_tensor.to(device)
            sentiment = sentiment.to(device)
            optimizer.zero_grad()

            #Initializing the hidden to 0
            hidden = net.init_hidden().to(device)

            #Passing the entire review through the model
            output, hidden = net(torch.unsqueeze(review_tensor[0],1), hidden)
            #input is of shape (num_words,1,300)
            #hidden is of shape (1,1,100)

            #Computing loss
            loss = criterion(output,sentiment)
            running_loss += loss.item()
            loop.set_postfix(loss=loss.item())
            loss.backward()
            optimizer.step()

        training_loss.append((running_loss)/(i+1))
        test_acc = testing(net,device,test_dataloader)
        test_accs.append(test_acc)
        net.train()
        print("[epoch: %d, batch: %5d] Loss: %.3f Test Accuracy: %.3f " % (epoch + 1, i


    return net, training_loss, test_accs
```

In [6]:

```
#Function to compute the confusion matrix
def confusion_matrix(model,test_data_loader,device):
  matrix = torch.zeros((2,2))
  model.eval()
  with torch.no_grad():
    for i,data in enumerate(test_dataloader):
       review_tensor,sentiment = data['review'], data['sentiment']
       review_tensor = review_tensor.to(device)
       sentiment = sentiment.to(device)
       #Initializing the hidden to 0
       hidden = model.init_hidden().to(device)

       #Passing the entire review through the model
       output, hidden = model(torch.unsqueeze(review_tensor[0],1), hidden)

       predicted = torch.max(output.data, 1)[1]
       matrix[predicted.cpu(),sentiment.cpu()] += 1

  heat = pd.DataFrame(matrix, index = [i for i in ["Negative","Positive"]],
                  columns = [i for i in ["Negative","Positive"]])
  heat = heat.astype(int)
  accuracy = (matrix.trace()/matrix.sum())*100
  plt.figure(figsize = (10,7))
  plt.title("Total accuracy is "+str(accuracy.item()))
  s = sn.heatmap(heat, annot=True,cmap='Blues',fmt='g')
  s.set(xlabel='Ground Truth', ylabel='Predicted')
```

In [7]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[7]:

```
device(type='cuda')
```

In [8]:

```
model = GRUnet(input_size=300,hidden_size=100,output_size=2,num_layers=2).to(device)
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,betas=(0.9,0.99))
epochs = 6
```

In [9]:

```
train_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/se
train_dataloader = torch.utils.data.DataLoader(train_dataset,batch_size=1,shuffle=True,

test_dataset = SentimentAnalysisDataset(dataset_file ="/content/drive/MyDrive/DL_HW8/ser
test_dataloader = torch.utils.data.DataLoader(test_dataset,batch_size=1,shuffle=True, nu

print("The size of the training dataset is "+ str(len(train_dataset)))
print("The size of the test dataset is "+ str(len(test_dataset)))
```

```
The size of the training dataset is 14227
The size of the test dataset is 3563
```

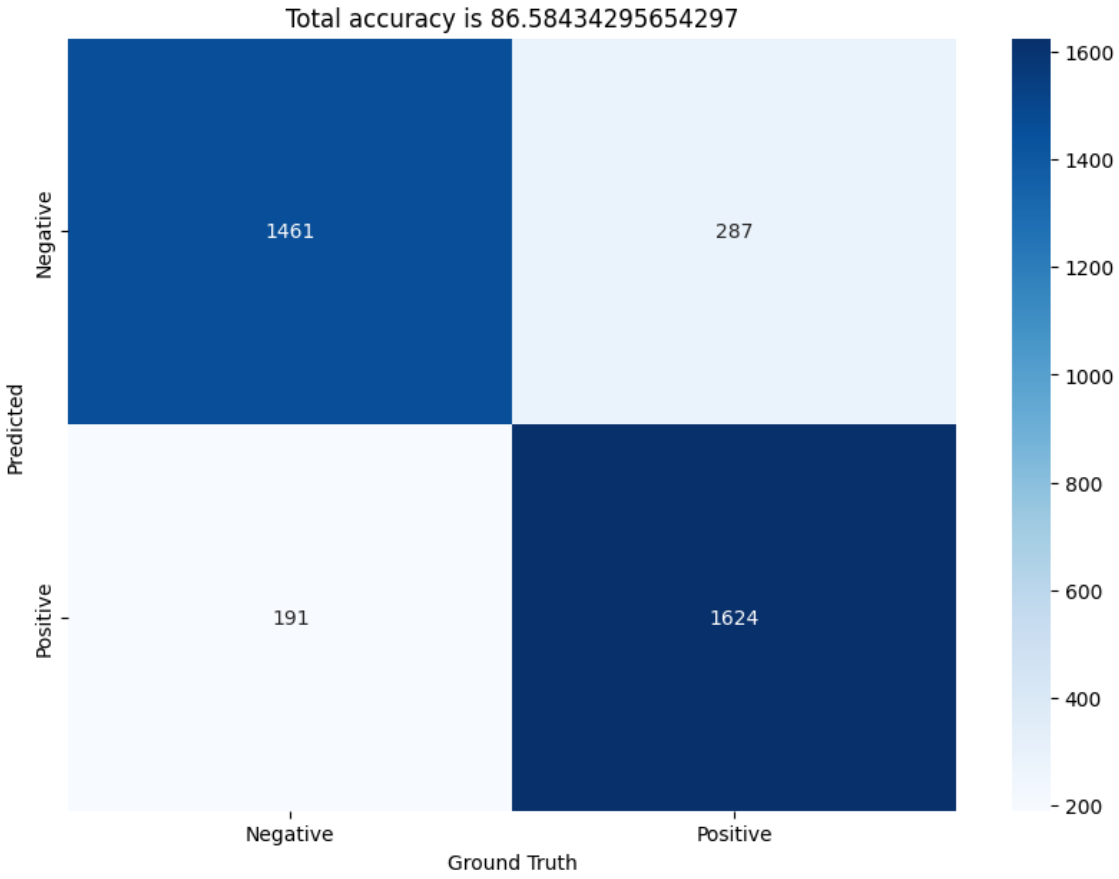
In [10]:

```
trained_model, training_loss, test_accs = training(model, criterion, optimizer, epochs,
```

```
100%|██████████| 14227/14227 [02:10<00:00, 108.74it/s, loss=0.431]
100%|██████████| 3563/3563 [00:13<00:00, 263.17it/s]

[epoch: 1, batch: 14227] Loss: 0.464 Test Accuracy: 83.188

100%|██████████| 14227/14227 [02:10<00:00, 109.23it/s, loss=0.1]
100%|██████████| 3563/3563 [00:13<00:00, 262.71it/s]

[epoch: 2, batch: 14227] Loss: 0.327 Test Accuracy: 87.118

100%|██████████| 14227/14227 [02:08<00:00, 111.15it/s, loss=0.6]
100%|██████████| 3563/3563 [00:13<00:00, 260.61it/s]

[epoch: 3, batch: 14227] Loss: 0.273 Test Accuracy: 84.199

100%|██████████| 14227/14227 [02:08<00:00, 110.62it/s, loss=0.00309]
100%|██████████| 3563/3563 [00:13<00:00, 254.54it/s]

[epoch: 4, batch: 14227] Loss: 0.218 Test Accuracy: 86.304

100%|██████████| 14227/14227 [02:08<00:00, 110.44it/s, loss=3.37e-5]
100%|██████████| 3563/3563 [00:13<00:00, 257.00it/s]

[epoch: 5, batch: 14227] Loss: 0.163 Test Accuracy: 87.567

100%|██████████| 14227/14227 [02:09<00:00, 110.17it/s, loss=1.19e-7]
100%|██████████| 3563/3563 [00:13<00:00, 261.36it/s]

[epoch: 6, batch: 14227] Loss: 0.129 Test Accuracy: 86.584
```

In [11]:

```
confusion_matrix(trained_model,test_dataloader,device)
```

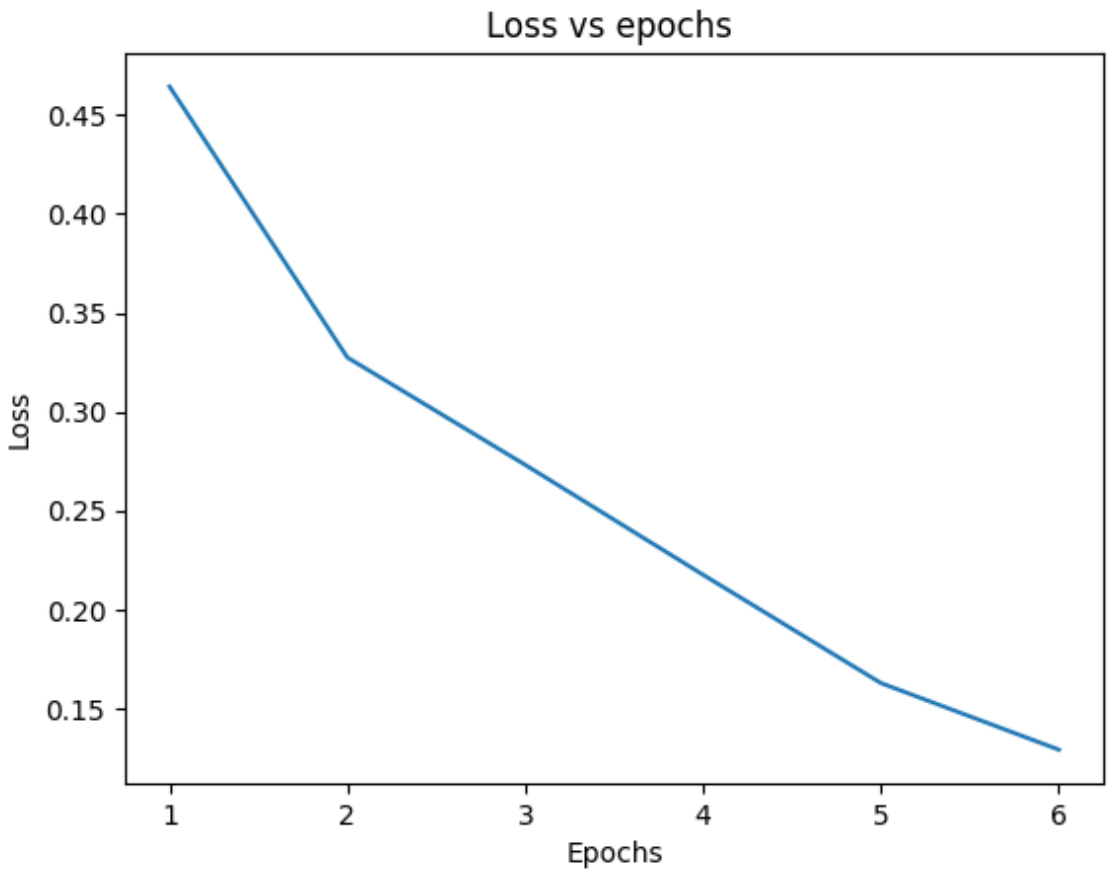


In [12]:

```
#Plotting the training loss vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs epochs ")
plt.plot(epochs,training_loss)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x7f3a135fb970>]
```

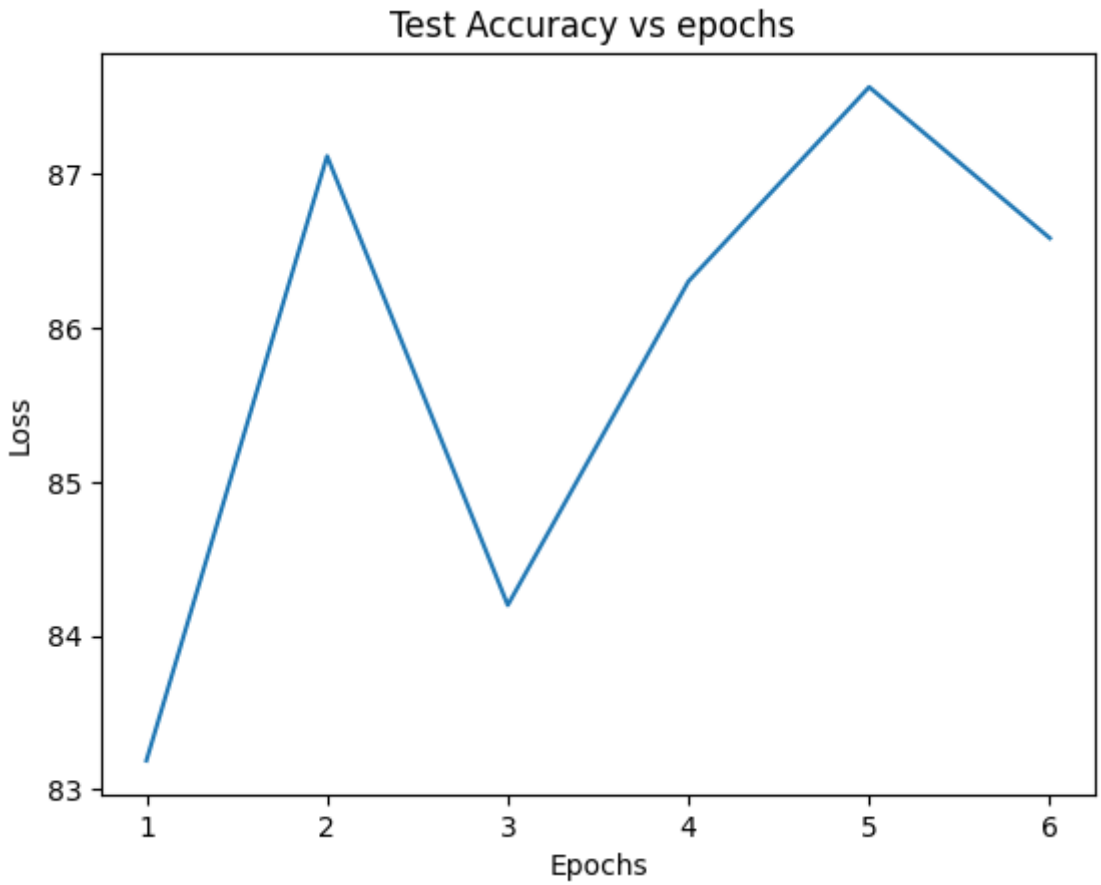


In [13]:

```
#Plotting the test accuracy vs epochs
epochs = np.arange(1,7)
plt.xticks(epochs, epochs)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Test Accuracy vs epochs ")
plt.plot(epochs,test_accs)
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x7f3a1357d4f0>]
```



In [14]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

In [15]:

```
#Number of parameters in the model
count_parameters(trained_model)
```

Out[15]:

```
422802
```

In [15]: