Sentiment Analysis of Yelp reviews using PyTorch Lightning and Torchtext

Author: Skerdi Ponde

Date: March 10, 2024

Project Overview

The aim of this project was to build a deep learning model that can perform sentiment analysis of yelp restraurant reviews. The reviews used in this project come from https://archive.ics.uci.edu/dataset/331/sentiment+labelled+sentences. This dataset was created for the Paper 'From Group to Individual Labels using Deep Features', Kotzias et. al,. KDD 2015. Each review is accompanied by a sentiment score: '1' for positive and '0' for negative.

PyTorch Lightning was used to build the sentiment analysis model. Lightning is a lightweight wrapper designed to make PyTorch projects more organized. It simplifies the process of developing, training and testing deep learning models by providing high-level abstractions and pre-defined functionalities.

Torchtext was used to prepare the dataset for the deep learning task. It simplifies the process of working with text data in natural language processing. Torchtext provides useful features such as tokenization (breaking text into separate words), creating a vocabulary of these words and mapping each of them to indices. This is necessary to convert the words to numerical representations (embeddings), which can then be used to train the deep learning model.

System information:

- Python Implementation CPython
- Python version: 3.11.4
- IPython version: 8.14.0
- torch: 2.0.1
- torchtext: 0.6.0
- lightning: 2.0.5

Importing the relevant modules

In [1]:
```
import numpy as np

import torch
from torch import nn
# import torch.nn.functional as F
# from torch.utils.data import TensorDataset, DataLoader
from torch.optim import Adam
from torch.optim.lr_scheduler import ReduceLROnPlateau

import torchtext
from torchtext.data.utils import get_tokenizer

import lightning as L
from lightning.pytorch.callbacks import EarlyStopping
from lightning.pytorch.loggers import TensorBoardLogger

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc
import matplotlib.pyplot as plt

import pandas as pd

import string
import re

import random
```

In [2]:
```
random_seed=21
random.seed(random_seed) # Use for splitting the dataset so the results are consistent

# Define the log directory where the logs will be saved
log_dir = "C:\\Users\\35569\\Desktop\\pytorch_sentiment_analysis\\new_logs"
```

Import the .txt file to pandas dataframe

The dataset is imported to a pandas dataframe from the path to the downloaded file. The reviews and their corresponding labels are seperated by tabs, hence we need to use the character '\t' as a separator in read_csv(). This is what it looks like.

In [3]:
```
df = pd.read_csv('sentiment labelled sentences\\yelp_labelled.txt', sep='\t', names=['review','sentiment'])
df.head()
```

Out[3]:

	review	sentiment
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

Prepare the Dataset

Preprocessing text

The raw data cannot be directly used as input to the model, so it has to be preprocessed. The following function removes punctuation and deals with the most common contractions of the English language.

In [4]:
```
def preprocess_text(text):

    # Deal with contractions
    text = re.sub(r"won't", "will not", text, flags=re.IGNORECASE)
    text = re.sub(r"'m", " am", text, flags=re.IGNORECASE)
    text = re.sub(r"n't", " not", text, flags=re.IGNORECASE)
    text = re.sub(r"'ve", " have", text, flags=re.IGNORECASE)
    text = re.sub(r"'d", " would", text, flags=re.IGNORECASE)
    text = re.sub(r"'ll", " will", text, flags=re.IGNORECASE)
    text = re.sub(r"'re", " are", text, flags=re.IGNORECASE)
    text = re.sub(r"'s", "", text)

    # Remove punctuation
    text = text.translate(str.maketrans(string.punctuation, " " * len(string.punctuation)))

    return text
```

The preprocessing function is applied to the review column of the pandas dataframe and this modified dataframe is saved in a new file. Notice the changes.

In [5]:
```
df['review'] = df['review'].apply(preprocess_text)
df['sentiment'] = df['sentiment'].astype(int)
df.to_csv("sentiment labelled sentences\\modified_yelp_labelled.txt", index=False)
df.head()
```

Out[5]:

	review	sentiment
0	Wow Loved this place	1
1	Crust is not good	0
2	Not tasty and the texture was just nasty	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

In [6]:
```
del df
```

Creating the Tabular Dataset

TabularDataset from torchtext is a convinient way to handle tabular data from .csv or .tsv files. First the input and label fields are defined as TEXT and LABEL through torchtext.data.Field and .LabelField. torchtext.data.Field allows us to further process the text by tokenizing it. What this means is that the text from each review is transfromed into a list of separate "words". There are different types of tokenizers. The 'basic_english' was chosen for this example since it can deal with consecutive spaces and converts all words to lowercase. This is important, because otherwise, e.g. the words "food" and "Food" would have different embeddings (explained later).

In [7]:
```
tokenizer = get_tokenizer('basic_english')
# Define the Fields for 'review' and 'sentiment' columns
TEXT = torchtext.data.Field(sequential=True, tokenize=tokenizer)
LABEL = torchtext.data.LabelField(dtype=torch.long)

# Create the TabularDataset
fields = [('review', TEXT), ('sentiment', LABEL)]
dataset = torchtext.data.TabularDataset(
    path="sentiment labelled sentences\\modified_yelp_labelled.txt", format='csv',
    fields=fields, skip_header=True
```

```
)

# Example
print(vars(dataset.examples[510]))
```

```
{'review': ['i', 'do', 'not', 'each', 'much', 'pasta', 'but', 'i', 'love', 'the', 'homemade', 'hand', 'made', 'pastas', 'and',
'thin', 'pizzas', 'here'], 'sentiment': '1'}
```

Splitting the dataset into training, validation and test data

The dataset is split into training, validation and test data by using the random_state from the generated seed. A seed is necessary, because otherwise the split of the dataset would be different and result of the training process would vary each time this piece of code was executed.

- **train_data** is used to train the model, i.e. learn the best parameters to perform the task.
- **val_data** is used to monitor how the model performs with unseen data during the training process.
- **test_data** is finally used to evaluate the accuracy of the model.

In [8]:
```
train_data, val_data, test_data = dataset.split(split_ratio=[0.7, 0.15, 0.15], random_state=random.getstate())

# Print the number of examples in each set
print("Number of training examples:", len(train_data))
print("Number of validation examples:", len(val_data))
print("Number of test examples:", len(test_data))
```

```
Number of training examples: 700
Number of validation examples: 150
Number of test examples: 150
```

Building the vocabulary

A vocabulary needs to be built for both TEXT and LABEL fields. That is esentially creating a dictionary that maps unique tokens to unique indices. These indices make it easier for the model to look up the embedding of each token.

As mentioned before, the deep learning model cannot directly process text, so it has to be converted to a numerical representation. Each review has to be converted to a sequence of tokens and each token has to be converted to a vector, such that numerical operations can be applied. A method you could think of is creating a vector with length equal to the number of words in your vocabulary and placing a 1 in the entry for that word and 0s in the others. This is called one-hot encoding, but it is not memory efficient (imaging having a vocabulary of millions of words) and it also holds no information about context, similarities between different words/tokens. Here is where embeddings come into play.

Word Embeddings

Word Embeddings are dense vector representations of words, which usually have a dimension in the hundreds, which is way less then the number of words that could be in the vocabulary. **Word2Vec** is one of the most known implementations of word embeddings. It consists of two approaches: **CBow** and **Skip-gram**. Both use one-hot encoded version of words in order to learn the embeddings. A sliding window selects groups of words. In the CBow algorithm the surrounding words are used as input to a shallow neural network to predict the middle word, while in Skip-gram the middle word is used to predict the surrounding words. In CBow the embeddings of a word are extracted from the weights which connect the hidden layer with the output prediction, while in Skip-gram they are extracted from the weights which connect the input to the hidden layer.

Since similar words are likely to be surrounded by the same neighboring words, the embedding model will eventually learn that similar words need to have similar embeddings (multiplied to similar weights) to produce similar results. That means that similar words will be close to each other in the embedding space. So, not only is this an efficient way to convert a word to a vector, but it also preserves their context.

In this project however, I used pretrained embeddings from **GloVe (Global Vectors)**, which is an extension of Word2Vec. Unlike Word2Vec, GloVe implements a word co-occurrence matrix by looking at the global conext instead of local. Dimensionality reduction of this matrix is preformed in order to learn the embeddings.

It is also possible to train your own embeddings, but it is not a good idea in this case considering that the size of the dataset is small.

In [9]:
```
TEXT.build_vocab(train_data, vectors = 'glove.6B.300d') # Text from Twitter, 6 billion tokens, 400k unique words, 300 dim embe
LABEL.build_vocab(train_data)
print(f"Text vocab size: {len(TEXT.vocab)}")
print(f"Label vocab size: {len(LABEL.vocab)}")
print(f"First 10 tokens in the vocab: {TEXT.vocab.itos[:10]}") # Index to string
# Example
word_to_check = 'food'
word_index = TEXT.vocab.stoi[word_to_check] # String to index
print(f"Index of word 'food': {word_index}")
```

```
Text vocab size: 1652
Label vocab size: 2
First 10 tokens in the vocab: ['<unk>', '<pad>', 'the', 'and', 'i', 'was', 'a', 'to', 'not', 'is']
Index of word 'food': 13
```

`<unk>` and `<pad>` are two additional tokens added when building the vocab.
`<unk>` is a token used for unseen words. It is by default initalized to a vector filled with zeros.
`<pad>` is a token used to match the dimension of sentences with different lengths.

Creating Dataloaders

In this section the dataloaders for training, validation and test data using BucketIterator from torchtext. The BucketIterator splits them in batches in an efficient way. Since a batch must have fixed dimensions, it groups similar length sentences together so that the padding required is minimal.

In [10]:
```
BATCH_SIZE = 20
train_loader, val_loader, test_loader = torchtext.data.BucketIterator.splits(
    (train_data, val_data, test_data),
    batch_size=BATCH_SIZE,
    sort_within_batch=True,
    sort_key=lambda x: len(x.review)
)
```

In the example below, the BucketIterator has grouped together in batches 20 sentences for different maximal lengths.

In [11]:
```
for batch in train_loader:
    print(f'Input matrix size: {batch.review.size()}')
    print(f'Output vector size: {batch.sentiment.size()}')
    break
```

```
Input matrix size: torch.Size([11, 20])
Output vector size: torch.Size([20])
```

Sentiment Analysis model

The "sentiment" model is built by inheriting from lighting.LightningModule. This would be the equivalent of inheriting from pytorch.nn.Module, if pure PyTorch was to be used. This model consists of 3 components:

- **Embedding Layer**, which converts sequences of tokens to sequences of vectors based on the pretrained embeddings from GloVe. This layer can be trained/fine-tuned, however in this case the freeze argument is set to True so that it doesn't lead to overfitting. I tried both freezing and letting the embedding layer train, however I did not notice any significant differences in the validation accuracy, besides a slight increase in train accuracy.
- **LSTM (Long-Short Term Memory)unit**, which is a form of RNN (recurrent neural network). RNNs are designed to work with input sequences of different lengths. They are unrolled as many times as the sequence is long and each input of the sequence is fed to each copy of the RNN. The output of the LSTM is a vector with hidden_dim entries. (That is for 1 data point)
- **Linear Layer**, which maps from the hidden dimension to the output dimension. The output dimension is 2, since this is a binary classification problem.

The goal is to maximize the probability of the correct class, or in other words minimize the loss. An appropriate loss function for this task is the **Cross Entropy Loss** (**nn.CrossEntropyLoss** in PyTorch). It is the sum of the negative logarithms of the "probabilities" for the correct class calculated over the batch. It is particularly useful because it has a really steep slope in the region close to 0 (very low propability predicted for the correct class). In PyTorch, the raw outputs (logits) can be passed to nn.CrossEntropyLoss directly, because it will automatically turn them into probabilities using the **Softmax** function and apply the logarithm before computing the loss.

Another common loss function for classification tasks is the **Negative Log Likelihood Loss** (**nn.NLLLoss** in PyTorch). This does the same thing as nn.CrossEntropyLoss, but it expects log probabilities as arguments, so, one has to apply these operations beforehand. According to the definition of nn.CrossEntropyLoss, it is a combination of nn.LogSoftmax and nnNLLLoss.

Cross Entropy Loss

$$CE = -\sum_{i=1}^{n} t_i \log(\sigma(\vec{x})_i)$$

$t_i =$ truth label for the i-th class
$\sigma(\vec{x})_i =$ softmax probability of logit vector $\vec{x}$ for i-th class

Softmax Function

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

$\sigma =$ softmax function
$\vec{x} =$ logit vector
$K =$ number of classes, dimension of logit vector
$e^{x_i} =$ exponential of i-th element in logit vector
$e^{x_j} =$ exponential of each element in logit vector

Softmax definition: https://en.wikipedia.org/wiki/Softmax_function
Cross Entropy Loss defintion: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

As mentioned earlier, lightning makes the process of developing deep learning networks easier. Instead of having to write a loop for training, update the parameters, empty the gradient and such ourselves, lightning provides us with the method training_step(), which simplifies the

process. It also includes logger functionalities. Logging the activity of important metrics such as the loss and accuracy, provides a better insight on what is happening during the training process. The progress graphs generated by the logger can be viewed in a UI such as tensorboard and can be compared to others to decide which hyperparameter selection is optimal for the given task.

Additionally, two other methods have been defined: validation_step() and test_step(). validation_step() is needed to understand how the model is performing on unseen data during the training process. A common issue with small-sized datasets is that **overfitting** may occur. Overfitting is when the model adapts really well to the training data, but does not generalize/perform well on unseen data. An indicator of this is when the training loss is decreasing, but the validation loss is increasing. By keeping track of validation metrics, we can stop the training process when noticing unwanted changes on the models behavior with unseen data. This is called **early stopping**.

test_step() is required to evaluate the model after the training process has finished.

In [12]:
```
class sentiment(L.LightningModule):
    def __init__(self, output_dim=2, embedding_dim=300, hidden_dim=200):

        super().__init__()

        self.embedding_layer = nn.Embedding.from_pretrained(TEXT.vocab.vectors, freeze=True)

        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim)

        self.fc = nn.Linear(in_features=hidden_dim, out_features=output_dim)

        self.loss_fn = nn.CrossEntropyLoss()

        self.learning_rate = 1.5e-3

        self.val_list=[]
        self.lr_list=[]

    def forward(self, input):
        # input is of dim (review length, batch_size), note: revie length refers
        # to the first dim of the batch or the length of the longest review

        embedded_text = self.embedding_layer(input)
        # embedded_text dim (review length, batch_size, embedding_dim)

        lstm_out, temp = self.lstm(embedded_text)
        lstm_out = lstm_out[-1]
        # lstm_out holds the hidden_state value at each time step so we retrieve the last
        # the final dim of lstm_out are (batch_size, hidden_dim)

        output = self.fc(lstm_out)
        # output dim (batch_size, output_dim)

        return output

    def configure_optimizers(self):

        optimizer = Adam(self.parameters(), lr=self.learning_rate)
        # optimizer = torch.optim.SGD(self.parameters(), lr=self.learning_rate) # Needs larger lr

        return optimizer

    def training_step(self, batch, batch_idx):

        input_i, label_i = batch
        output_i = self.forward(input_i)
        output_probs_i = torch.softmax(output_i, dim=1)
        pred_i = torch.argmax(output_probs_i, dim=1)

        label_i = label_i.view(-1).to(torch.long)

        accuracy = torch.sum(label_i==pred_i)/len(label_i)

        loss = self.loss_fn(output_probs_i, label_i)

        self.log("train_accuracy", accuracy, on_step=False,
                 on_epoch = True, prog_bar=True, batch_size=BATCH_SIZE)

        # print(loss)

        self.log("train_loss", loss, batch_size=BATCH_SIZE)

        return loss

    def validation_step(self, batch, batch_idx):

        input_i, label_i = batch
        output_i = self.forward(input_i)
        output_probs_i = torch.softmax(output_i, dim=1)
        pred_i = torch.argmax(output_probs_i, dim=1)

        label_i = label_i.view(-1).to(torch.long)
```

```
        accuracy = torch.sum(label_i==pred_i)/len(label_i)

        loss = self.loss_fn(output_probs_i, label_i)

        self.log("val_accuracy", accuracy, on_step=False,
                 on_epoch = True, prog_bar=True, batch_size=BATCH_SIZE)
        self.log("val_loss", loss, batch_size=BATCH_SIZE, prog_bar=True)

        return loss

    def test_step(self, batch, batch_idx):

        input_i, label_i = batch
        output_i = self.forward(input_i)
        output_probs_i = torch.softmax(output_i, dim=1)
        pred_i = torch.argmax(output_probs_i, dim=1)

        label_i = label_i.view(-1).to(torch.long)

        accuracy = torch.sum(label_i==pred_i)/len(label_i)

        loss = self.loss_fn(output_probs_i, label_i)

        metrics = {"test_accuracy": accuracy, "test_loss": loss}
        self.log_dict(metrics, batch_size=BATCH_SIZE)
        return metrics
```

Training Process

Creating trainer object

Firstly, create a sentiment() object called model. Defining an EarlyStopping callback which monitors val_loss is optional. However, it can be potentially useful because it will stop the training process when the monitored metric stops decreasing. It deals with overfitting and also saves time when training on large datasets. The lightning.Trainer object is needed to train our model.

In [13]:
```
torch.manual_seed(random_seed) # Same parameter initialization each time
model=sentiment()
early_stop_callback = EarlyStopping(
            monitor='val_loss',  # The metric to monitor for early stopping
            patience=40,  # Number of epochs with no improvement after which training will be stopped
            stopping_threshold=0.10, # Immediately stop training when the metric reaches this threshold
            mode='min',  # 'max' mode means the early stop is applied when the monitored metric stops decreasing
            min_delta=0.01,  # Amount of change in order to count it as an improvement
            verbose=False
        )

# Instantiate the logger (this is so we can take a look at the training process later)
logger = TensorBoardLogger(save_dir=log_dir)
# Remove/Add this argument: callbacks=[early_stop_callback] to enable/disable early stopping
trainer = L.Trainer(max_epochs=100, log_every_n_steps=2, callbacks=[early_stop_callback], logger=logger)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

Learning rate finder

This next step is also optional. A Tuner object can be used to find an optimal learning rate for the training process. A relatively large learning rate will lead to our model's parameters bouncing around and possibly never converge to the global minimum, while a small one might take too long to converge to the minimum or get stuck in local minima. So finding a good learning rate is crucial for training. The method lr_find() tests different learning rate values within the chosen range and provides a suggestion. This suggestion is valid only for the initial epochs of the training process because in many cases it may not work so well afterwards. Updating the learning rate during training may yield better results, e.g. starting with a large learning rate so it converges fast towards the minimum and steadily decreasing it so it doesn't bounce around it. This can be done using a learning rate scheduler, but I am not going to include that here because I didn't notice significant differences in training.

In [15]:
```
tuner = L.pytorch.tuner.tuning.Tuner(trainer)

lr_find_results = tuner.lr_find(model,
                                train_dataloaders=train_loader,
                                val_dataloaders=val_loader,
                                num_training= 1000, # Number of points to test
                                min_lr=1e-5, # Minimum of the range of values
                                max_lr=1e-1, # Maximum
                                mode = 'exponential',
                                early_stop_threshold=None, # Threshold to stop the search
                                update_attr=True # Automatically updates lr
                               )
```

```
# lr_find_results.results
# print(vars(lr_find_results))
```

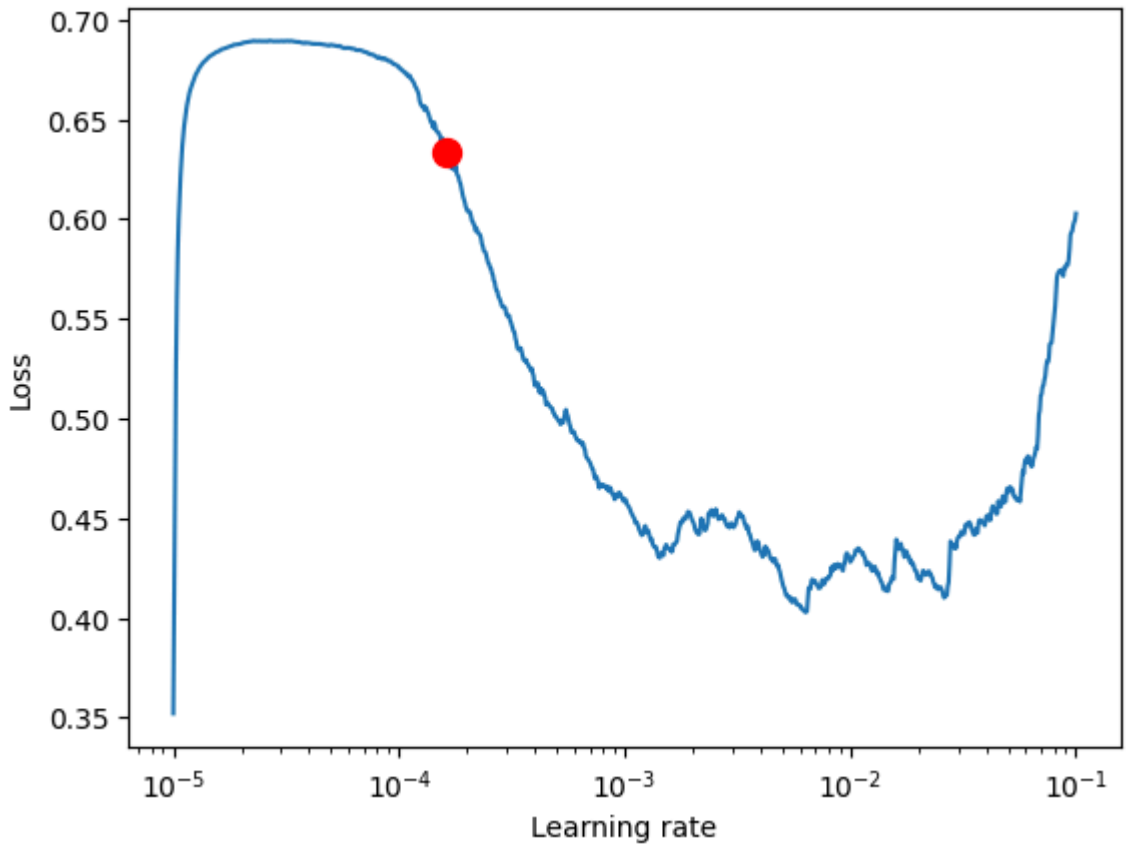
```
Finding best initial lr: 100%|██████████| 999/1000 [00:25<00:00,  8.42it/s]`Trainer.fit` stopped: `max_steps=1000` reached.
Finding best initial lr: 100%|██████████| 1000/1000 [00:25<00:00, 39.03it/s]
Learning rate set to 0.00016143585568264863
Restoring states from the checkpoint path at c:\Users\35569\Desktop\pytorch_sentiment_analysis\.lr_find_32f2164e-02d1-4e92-932a
-1fd04342f0a2.ckpt
Restored all states from the checkpoint at c:\Users\35569\Desktop\pytorch_sentiment_analysis\.lr_find_32f2164e-02d1-4e92-932a-1
fd04342f0a2.ckpt
```

Something to note is that the lr_find() method would come up with different suggestions each separate run, even though the random seed was fixed. I am not sure why this would occur and could not find any discussions online regarding this issue. Maybe there is some other randomness that cannot be fixed by random.seed() and torch.manual_seed().

Despite that, the curve of the dependency of the loss with respect to the initial learning rate can be plotted. According to the documentation, a good initial learning rate should be somewhere in the middle of the steepest decreasing slope and not at a local minimum. If this is not the case for the produced graph, you can choose your own following this criteria.

In [16]:
```
fig = lr_find_results.plot(suggest=True)

# Get the suggested learning rate
suggested_lr = lr_find_results.suggestion()

# Print the suggested learning rate
print("Suggested Learning Rate:", suggested_lr)
```

```
Suggested Learning Rate: 0.00016143585568264863
```



In [17]:
```
trainer.fit(model, train_dataloaders=train_loader, val_dataloaders=val_loader)
```

```
  | Name            | Type             | Params
-------------------------------------------------------
0 | embedding_layer | Embedding        | 495 K
1 | lstm            | LSTM             | 401 K
2 | fc              | Linear           | 402
3 | loss_fn         | CrossEntropyLoss | 0
-------------------------------------------------------
402 K     Trainable params
495 K     Non-trainable params
897 K     Total params
3.590     Total estimated model params size (MB)
Epoch 60: 100%|██████████| 35/35 [00:01<00:00, 32.37it/s, v_num=0, val_accuracy=0.806, val_loss=0.507, train_accuracy=0.977]
```

Storing and loading the trained weights

The weights can be saved in a .pth file after training, so they can be loaded later to be used for inference or further training.

In [18]:
```
filepath = 'ce_hd_200_weights.pth'
torch.save(model.state_dict(), filepath)
```

Here's how model weights are loaded

In [ ]:
```
# model.load_state_dict(torch.load('path/to/model_weights.pth'))
# model=sentiment()
# model.load_state_dict(torch.load('ce_hd_200_weights.pth'))
```

When training is finished, Lightning will also store a checkpoint of the process inside of the defined log_dir. This can be accessed through trainer.checkpoint_callback.best_model_path

In [19]:
```
path_to_best_checkpoint = trainer.checkpoint_callback.best_model_path
print(path_to_best_checkpoint)
```

```
C:\Users\35569\Desktop\pytorch_sentiment_analysis\new_logs\lightning_logs\version_0\checkpoints\epoch=60-step=2135.ckpt
```

Continue training from the checkpoint of the best model path. Additionally, you can store your own checkpoints during the training process. For that refer to this link.

In [20]:
```
trainer = L.Trainer(max_epochs=65, log_every_n_steps=2, logger=logger) # Note: max_epoch should be larger than current_epoch
trainer.fit(model, train_dataloaders=train_loader, val_dataloaders = val_loader,
            ckpt_path=path_to_best_checkpoint)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
c:\Users\35569\Desktop\pytorch_sentiment_analysis\myenv\Lib\site-packages\lightning\pytorch\callbacks\model_checkpoint.py:615:
UserWarning: Checkpoint directory C:\Users\35569\Desktop\pytorch_sentiment_analysis\new_logs\lightning_logs\version_0\checkpoin
ts exists and is not empty.
  rank_zero_warn(f"Checkpoint directory {dirpath} exists and is not empty.")
Restoring states from the checkpoint path at C:\Users\35569\Desktop\pytorch_sentiment_analysis\new_logs\lightning_logs\version_
0\checkpoints\epoch=60-step=2135.ckpt
c:\Users\35569\Desktop\pytorch_sentiment_analysis\myenv\Lib\site-packages\lightning\pytorch\trainer\call.py:247: UserWarning: B
e aware that when using `ckpt_path`, callbacks used to create the checkpoint need to be provided during `Trainer` instantiatio
n. Please add the following callbacks: ["EarlyStopping{'monitor': 'val_loss', 'mode': 'min'}"].
  rank_zero_warn(

  | Name            | Type             | Params
-----------------------------------------------------
0 | embedding_layer | Embedding        | 495 K
1 | lstm            | LSTM             | 401 K
2 | fc              | Linear           | 402
3 | loss_fn         | CrossEntropyLoss | 0
-----------------------------------------------------
402 K     Trainable params
495 K     Non-trainable params
897 K     Total params
3.590     Total estimated model params size (MB)
Restored all states from the checkpoint at C:\Users\35569\Desktop\pytorch_sentiment_analysis\new_logs\lightning_logs\version_0
\checkpoints\epoch=60-step=2135.ckpt
Epoch 64: 100%|██████████| 35/35 [00:00<00:00, 39.09it/s, v_num=0, val_accuracy=0.794, val_loss=0.522, train_accuracy=0.976]
`Trainer.fit` stopped: `max_epochs=65` reached.
Epoch 64: 100%|██████████| 35/35 [00:00<00:00, 37.42it/s, v_num=0, val_accuracy=0.794, val_loss=0.522, train_accuracy=0.976]
```

Tensorboard - Monitoring the training process

As mentioned earlier, the progress of performance metrics of the model during the training process are logged through PyTorch Lightning. These metrics can be viewed with Tensorboard.

TensorBoard is a powerful visualization tool that is compatible with TensorFlow and PyTorch. It provides a suite of tools for visualizing and analyzing various aspects of machine learning models. Its primary uses include:

- **Monitoring Training Metrics**: Visualize metrics like loss and accuracy during training and validation to understand how well the model is learning.
- **Visualizing Graphs**: See the computational graph of the model to understand its structure and operations.
- **Analyzing Distributions**: Track distributions of weights and biases to observe how they evolve during training.
- **Comparing Runs**: Compare different runs of experiments to evaluate and choose the best-performing model configurations.

Usage in This Project

In this sentiment analysis project, TensorBoard is used to monitor and visualize the following metrics:

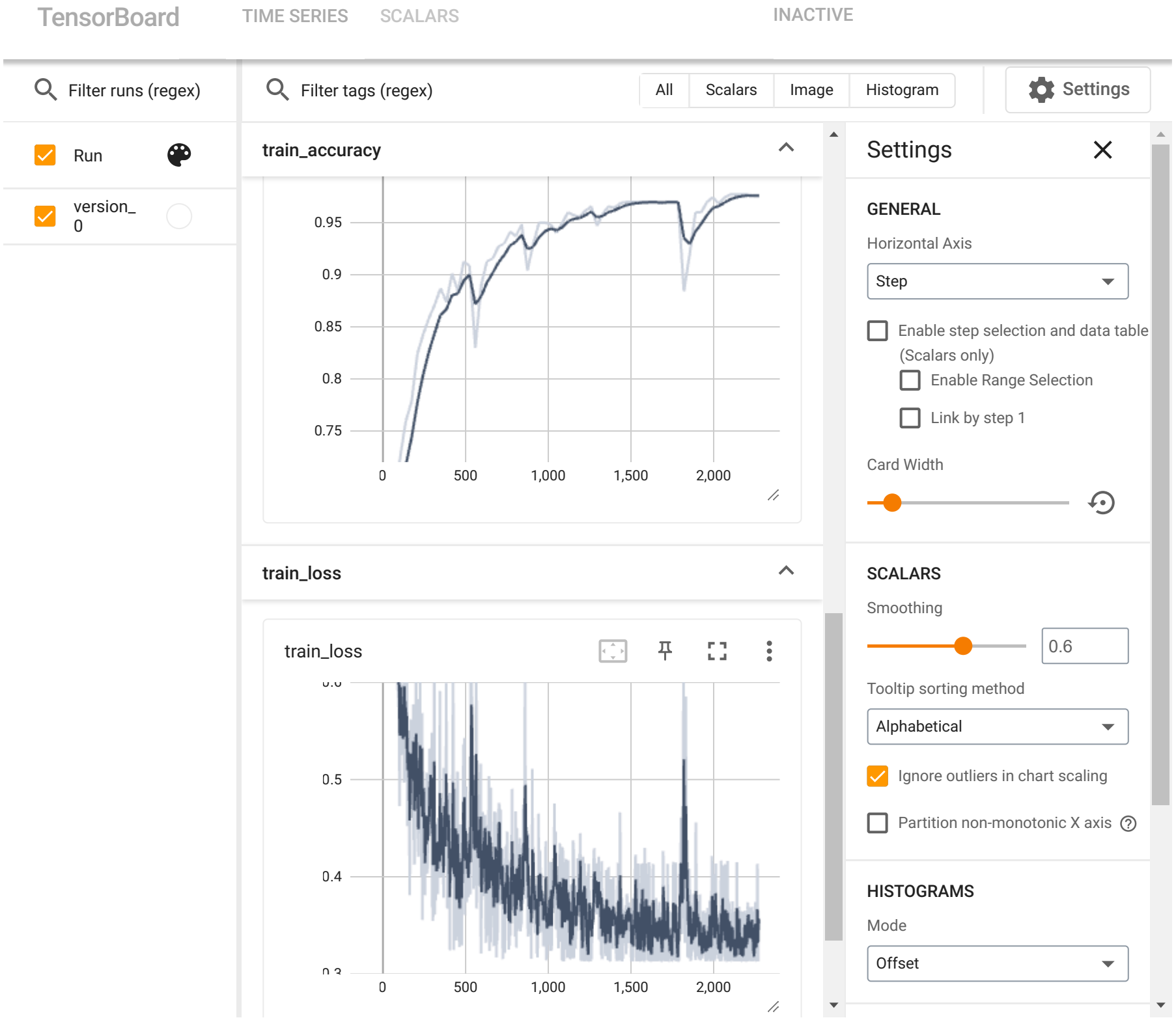
- **Training Loss**: The loss value calculated on the training dataset, indicating how well the model fits the training data.
- **Validation Loss**: The loss value calculated on the validation dataset, showing how well the model generalizes to unseen data.
- **Training Accuracy**: The accuracy of the model on the training dataset, reflecting its performance on the data it was trained on.
- **Validation Accuracy**: The accuracy of the model on the validation dataset, which helps assess the model's ability to generalize to new, unseen data.

The visualizations help in tracking the training progress and diagnosing potential issues like overfitting or underfitting.

In [27]:
```
%reload_ext tensorboard
```

In [28]:
```
from tensorboard import notebook
notebook.start("--logdir C:\\Users\\35569\\Desktop\\pytorch_sentiment_analysis\\new_logs\\lightning_logs --port 6006")
# You can also try executing the following command to launch a tensorboard session, however it wasnt displaying the <IPython.c
# %tensorboard --logdir "C:\\Users\\35569\\Desktop\\pytorch_sentiment_analysis\\new_logs\\lightning_logs" --port 6006
```

```
Reusing TensorBoard on port 6006 (pid 4648), started 0:02:08 ago. (Use '!kill 4648' to kill it.)
```



Testing the trained model

Define a function called **sentiment_analysis** that uses the trained model to classify a piece of text (string). It will return the class prediction and the associated probability.

In [29]:
```
def sentiment_analysis(text):

    modified = preprocess_text(text)
    tokenized = tokenizer(modified)
    indexed = torch.tensor(([TEXT.vocab.stoi[word] for word in tokenized]))
    # print(model(indexed)) # logits
    probabilities = torch.softmax(model(indexed).detach(), dim=0)
    prediction = torch.argmax(probabilities, 0).item()

    return prediction, probabilities

# Some of these may get incorrectly classified
reviews = ["The food took forever to prepare and the waiter was extremely rude to us.\
    Also the taste was mediocre so it wasn't worth the wait.",

    "I was very impressed. We only had to wait 5 minutes for the appetizer and \
    the rest of the food was also served really quickly. This is the best fish restaurant in town.",

    "The food was good",

    "The waiter was slow as heck.",

    "The pasta was bad and the pizza was undercooked."]

# When creating the tabular dataset from the TEXT and LABEL fields, the sentiment is saved as a string '0' for negative and '1
# Afterwards the the dataset is partitioned into train, validation and test data and then the vocabulary is build from the tra
# Based on the random seed and how the data is split, sometimes the label '0' may be mapped to the integer 1 and '1' to 0. In
# display results for the examples, we need to know this information.

print(LABEL.vocab.stoi)
```

```
# Check label mappings
label_mapping = LABEL.vocab.stoi
negative = label_mapping['0']
positive = label_mapping['1']


for i in range(len(reviews)):
    predicted_label, probabilities = sentiment_analysis(reviews[i])
    if predicted_label == negative:
        print(f"Review {i+1} is predicted as negative review with certainty {probabilities[negative]:.1%}")
    else:
        print(f"Review {i+1} is predicted as positive review with certainty {probabilities[positive]:.1%}")
```

```
defaultdict(None, {'1': 0, '0': 1})
Review 1 is predicted as negative review with certainty 100.0%
Review 2 is predicted as negative review with certainty 100.0%
Review 3 is predicted as positive review with certainty 97.6%
Review 4 is predicted as negative review with certainty 99.9%
Review 5 is predicted as negative review with certainty 100.0%
```

Performance metrics on test data

You can view the results for the test set as well.

In [30]:
```
test_results=trainer.test(model, dataloaders=test_loader)
```

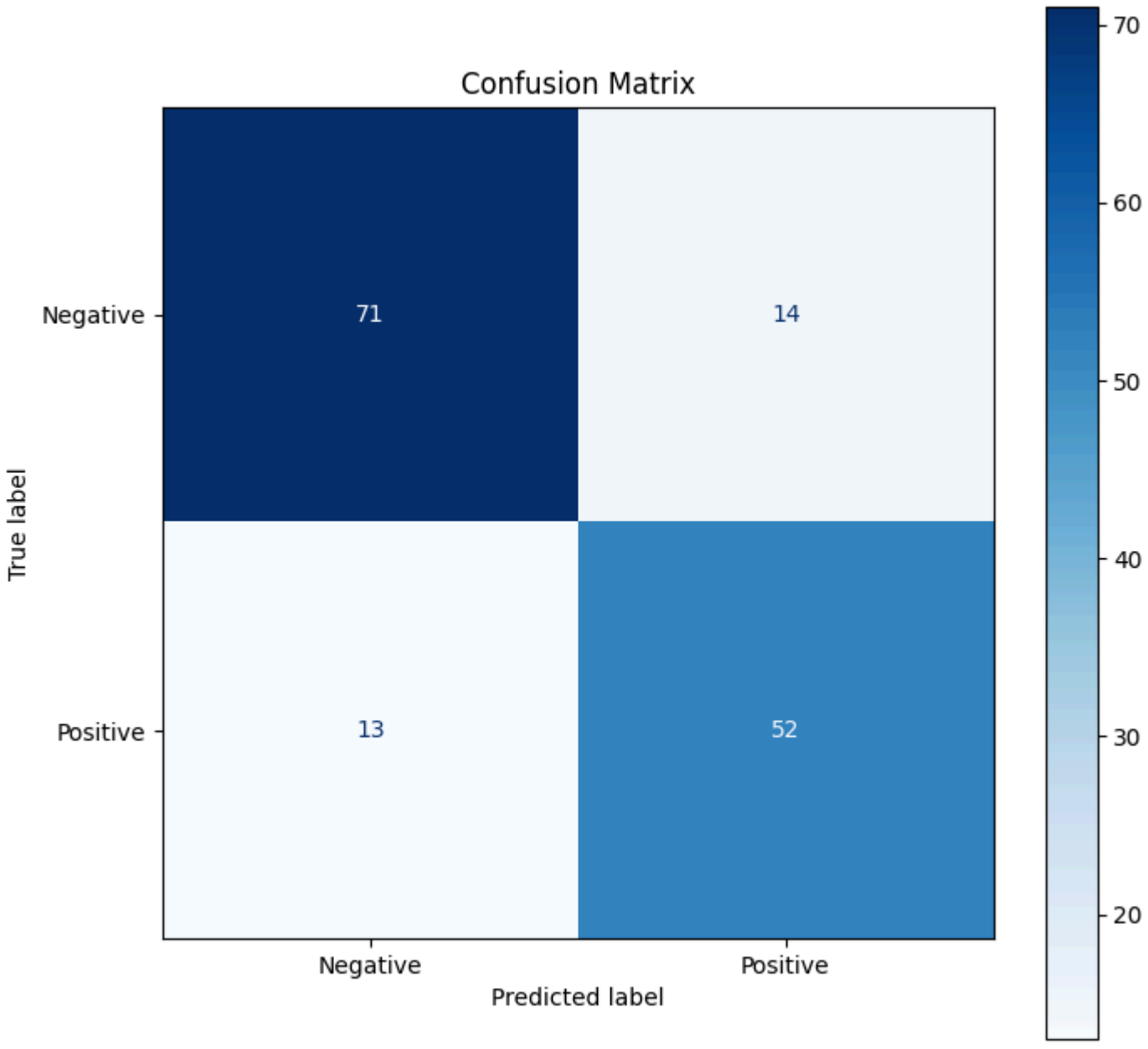
```
Testing DataLoader 0: 100%|██████████| 8/8 [00:00<00:00, 69.01it/s]
```

Runningstage.testing metric	DataLoader 0
test_accuracy	0.831250011920929
test_loss	0.47770509123802185

Let us also take a look at the confusion matrix of the trained model on test data. For this I will use the **confusion_matrix** function and **ConfusionMatrixDisplay** class from the library **scikit-learn**.

The default threshold value here is obviously 0.5.

In [31]:
```
# Set model to evaluation mode
model.eval()

# Collect predictions and true labels
all_preds = []
all_labels = []
all_outputs = []

# Check the corresponding labels and predictions of each input and append the lists
with torch.no_grad():
    for batch in test_loader:
        inputs, labels = batch.review, batch.sentiment
        outputs = model(inputs)
        probabilities = torch.softmax(outputs, dim=1).cpu().numpy()
        all_outputs.extend(probabilities)
        _, preds = torch.max(outputs, dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())


# Compute confusion matrix
cm = confusion_matrix(all_labels, all_preds, labels=[negative, positive])  # Binary classification
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Negative', 'Positive'])

# Plot confusion matrix
fig, ax = plt.subplots(figsize=(8, 8))
disp.plot(ax=ax, cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```



For completeness the **ROC curve** (Receiver Operating Characteristic Curve) is plotted. The ROC curve is used to evaluate the performance of a binary classification model. The **True Positive Rate (TPR = True Positive (predicted)/ Actual Positive)** is plotted against the **False Positive Rate (FPR = False Positive/ Actual Negative)** over various threshold values. The ROC curve helps visualize the trade-off between **Sensitivity (TPR)** and **Specificity (TNR = True Negative/ Actual Negative = 1 - FPR)**.

AUC is the area under the ROC curve. Generally, the higher the AUC, the better the model is at distinguishing between positive and negative cases. When AUC = 1 it indicates that the model is perfect, 0.9 - 1 indicates an excellent performance, 0.8 - 0.9 a good performance, 0.7 - 0.8 a fair performance, 0.5 - 0.7 a poor performance, while 0.5 indicates that the model is not doing better than just random guessing. A value below 0.5 on the other hand, indicates the model is predicting the incorrect class most of the time, which probably stems from an issue with the model configuration.

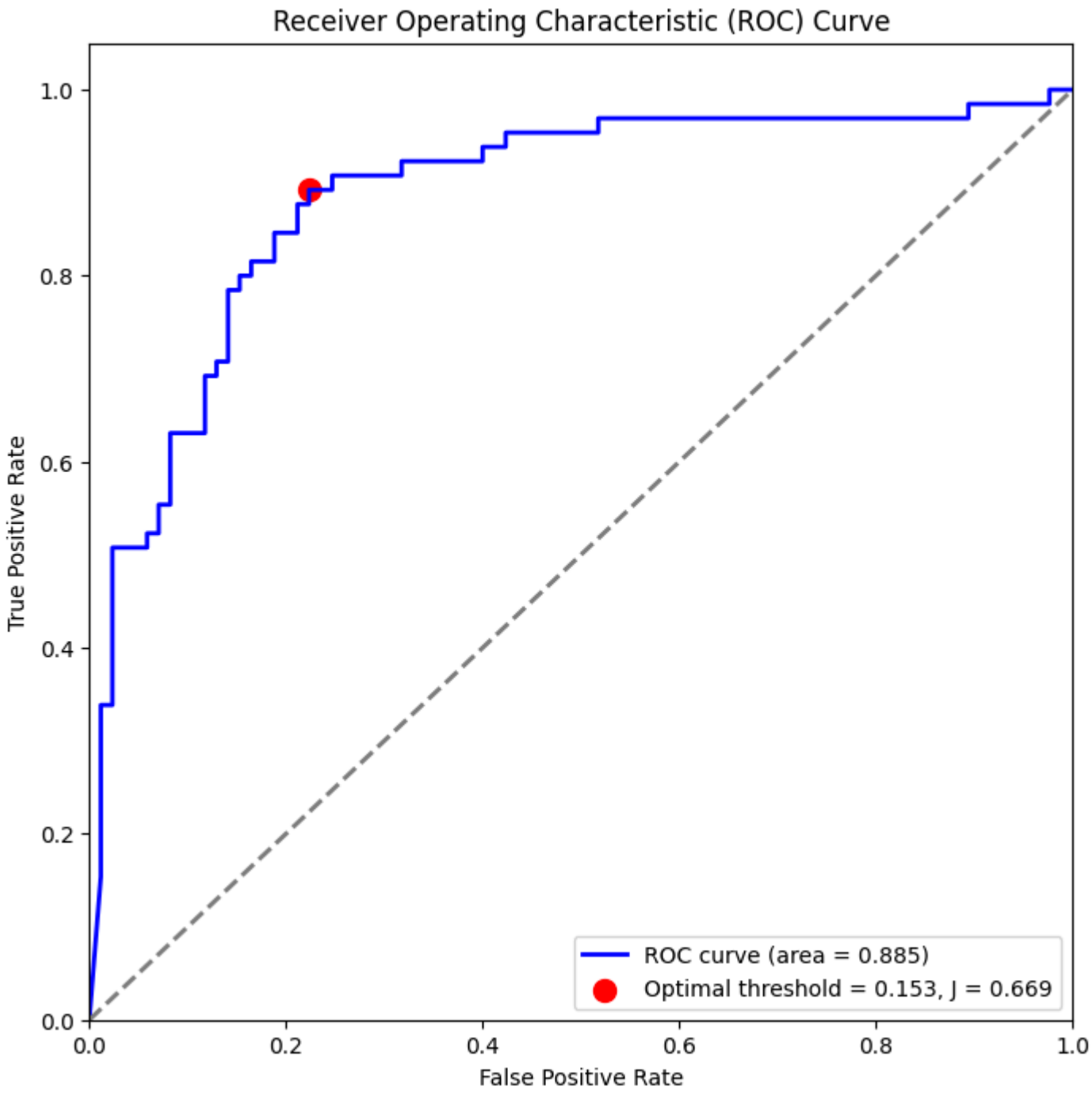
Another metric one might want to look at is **Youden's J statistic** or **Youden's Index** which is defined as **J = TPR + TNR - 1 = TPR - FPR**. We want to find the threshold which maximizes J (True Positive and True Negative Rates) for that particular model, which corresponds to the point that is the closest to the top-left corner. Here the TPR is highest, while FPR is the lowest.

To compute the ROC curve and AUC the functions **roc_curve** and **auc** for scikit-learn are employed. For this function to plot the ROC curve, it needs to be provided with the true labels of the data, the probabilities of the positive class and the label of the positive class (as mentioned earlier sometimes '1' - positive is mapped to a 0).

In [32]:
```
# Convert lists to numpy arrays
all_outputs = np.array(all_outputs)
all_labels = np.array(all_labels)

# Compute ROC curve and ROC area for the positive class
fpr, tpr, thresholds = roc_curve(all_labels, all_outputs[:, positive], pos_label=positive)
roc_auc = auc(fpr, tpr)

# Youden's J statistic = TPR + TNR - 1
# TNR = 1 - FPR
youden_j = tpr - fpr
best_threshold_index = np.argmax(youden_j)
best_threshold = thresholds[best_threshold_index]
optimal_tpr = tpr[best_threshold_index]
optimal_fpr = fpr[best_threshold_index]
print(optimal_tpr, optimal_fpr)

# Plot ROC curve
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.3f})')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.scatter(optimal_fpr, optimal_tpr, color='red', s=100,
            label=f'Optimal threshold = {best_threshold:.3f}, J = {(optimal_tpr - optimal_fpr):.3f}')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

```
0.8923076923076924 0.2235294117647059
```



When comparing the performance of different models to decide which is the best one for the given task, the aformentioned metrics such as ROC/AUC, accuracy, Youden's J statistic are observed. Other important metrics are **Precision = True Positive/ Predicted Positive** and **Recall = True Positive/ Actual Positive** which is the same as TPR and plotting the **Precision-Recall curve** for different thresholds. The optimal threshold in this case would be the one that maximizes the **F1-score**, which is the harmonic mean of Precision and Recall. These are particularly useful when the test set is imbalanced i.e. the number of positive is larger than negative or the opposite, however I have not included them in this analysis. Another thing to note is that the choice of the model/threshold also depends on the specific use case. For example, assume a model has been trained to identify persons who have a contagious disease based on certain symptoms or features and is being tested on a test set. In this scenario, we would much rather have false positives than false negatives. Although the computed optimal threshold for the ROC curve, for example, strikes a good balance between TPR and FPR, we would prefer picking a lower threshold, which might lower the number of false negatives.

Conclusion

This project demonstrates the effectivenes of deep learning for sentiment analysis using PyTorch Lightning and Torchtext. While the model effectively classified Yelp reviews as positive or negative, its performance could benefit significantly from a larger dataset, as the current sample size of 1,000 may limit its generalization capabilities. Additionally, employing alternative word embedding methods, such as subword embeddings, could help handle unseen tokens not included in the pre-trained embeddings.

Something to note is the observed overfitting, indicated by high accuracy on the training set but lower on the validation set. This suggests that the model successfully captures data patterns, however methods to mitigate overfitting could improve its performance. Implementing techniques like dropout layers or batch normalization could enhance the model's robustness.

Furthermore, exploring more advanced architectures, such as Transformers (e.g., BERT), may offer improved context understanding and sentiment classification.

References

- Kotzias, D., et al. (2015). From Group to Individual Labels using Deep Features. KDD 2015.

- L15.7 An RNN Sentiment Classifier in PyTorch by Sebastian Raschka
- PyTorch Lightning Documentation: https://pytorch-lightning.readthedocs.io
- Torchtext Documentation: https://pytorch.org/text/stable/index.html