# Final Project Report

Project Title: Sentiment analysis with RNN.

Course Name: CS 551 – Deep Learning

Professor's Name: The name of the professor or instructor.

University Name: Bishop’s University.

Students:

Johan Stevenson Mogollon Arcos / Student ID: 002359844

Fabio Cozzuto / Student ID: 002214965.

## Abstract

## In this article we report the results of the final project of the Deep Learning course, which used recurrent neural networks (RNN) and sequential data, in this case the IMDB dataset. We describe the steps that were followed to arrive at the final model, including data selection and preparation, RNN model application (focusing on LSTM and GRU alternatives), hyperparameter tuning (using Optuna), training and testing. The objective of this work was to gain practical experience with recurrent neural networks (RNN) using PyTorch, by implementing and training a model to help perform a given task. At the end of this exercise, to understand the practical applications of these models and in general to acquire skills that will be useful for possible future projects.

## Introduction

This report summarizes the work done to develop and evaluate a recurrent neural network (RNN) model created for sentiment analysis of the IMDB dataset. Sentiment analysis is a subfield of Natural Language Processing (NLP), which consists in determining the emotional tone or attitude expressed in a text, thus having numerous applications in everyday life.

The function of the RNN model is to classify the comments, as positive or negative, received by the movies found on the IMDB page. Thus, important aspects of the project include data preprocessing, model architecture design (LSTM and GRU variants), hyperparameter optimization, and finally model evaluation.

The challenge of this project is to use RNNs well for sentiment analysis. Since RNNs have been shown to be effective in different tasks involving sequences, care must be taken in configuring aspects such as data preprocessing, model design, and training processes. In addition, being able to reproduce the results in different configurations and models can be difficult due to disparities in data partitioning or the parameters used to initialize the model.

Now, sentiment analysis can provide valuable information to companies, researchers and individuals. Therefore, its applications can be varied in different industries. It is for this reason that the importance of the project to obtain the necessary skills to understand, implement and evaluate RNN models makes sense.

Given the above context, the key points of this project are in first place, to develop a clear and reproducible workflow for sentiment analysis using RNNs. Next, to implement and to compare the performance of LSTM and GRU based RNN models on the IMDB dataset. Also, to optimize the model hyperparameters, to evaluate the trained models using appropriate metrics. Finally, to document in detail the entire process, from data preprocessing to model evaluation, to ensure reproducibility.

## Dataset Selection and Preprocessing steps

* Dataset: IMDB review dataset

The dataset selected for this project is the IMDB dataset, this one is a text binary sentiment classification dataset, a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. We obtained this from TensorFlow Datasets, and we choose this dataset for text classification because of its popularity. It is a dataset with a good balance between positive and negative reviews, this give us a good dataset for training and evaluating a sentiment analysis model, but at the same time, an easy one to learn about RNN while we are creating a model for practical applications.

* Loading the data

We used the next function to load the data, and after loading we split it into three parts (Train, Validation and Test)

import tensorflow\_datasets as tfds

def load\_imdb\_data():

    imdb, \_ = tfds.load("imdb\_reviews", with\_info=True, as\_supervised=True)

    return imdb['train'], imdb['test']

train\_data, test\_data = load\_imdb\_data()

def split\_train\_validation(train\_data, validation\_split=0.2):

    train\_list = list(train\_data.skip(int(validation\_split \* len(list(train\_data)))).as\_numpy\_iterator())

    validation\_list = list(train\_data.take(int(validation\_split \* len(list(train\_data)))).as\_numpy\_iterator())

    train\_labels = [label for \_, label in train\_list]

    validation\_labels = [label for \_, label in validation\_list]

    print(f"Training set size: {len(train\_list)}")

    print(f"Validation set size: {len(validation\_list)}")

    return train\_list, validation\_list, train\_labels, validation\_labels

train\_list, validation\_list, train\_labels, validation\_labels = split\_train\_validation(train\_data)

test\_list = list(test\_data.as\_numpy\_iterator())

test\_labels = [label for \_, label in test\_list]

print(f"Test set size: {len(test\_list)}")

* Data Preprocessing

In the following code we implemented the preprocessing steps:

def remove\_html\_tags(text):

    """Removes HTML tags from a string."""

    clean = re.compile('<.\*?>')

    return re.sub(clean, '', text)

def decode\_clean\_text(data\_list):

    """Decodes byte strings and removes HTML tags from text data."""

    sentences = [remove\_html\_tags(text.decode('utf-8')) for text, \_ in data\_list]

    return sentences

def tokenize\_pad\_sequences(train\_sentences, validation\_sentences, test\_sentences, num\_words=20000, max\_length=200):

    """Tokenizes and pads text sequences."""

    tokenizer = Tokenizer(num\_words=num\_words, oov\_token="<OOV>")

    tokenizer.fit\_on\_texts(train\_sentences)

    train\_sequences = tokenizer.texts\_to\_sequences(train\_sentences)

    validation\_sequences = tokenizer.texts\_to\_sequences(validation\_sentences)

    test\_sequences = tokenizer.texts\_to\_sequences(test\_sentences)

    train\_padded = pad\_sequences(train\_sequences, maxlen=max\_length, padding='post', truncating='post')

    validation\_padded = pad\_sequences(validation\_sequences, maxlen=max\_length, padding='post', truncating='post')

    test\_padded = pad\_sequences(test\_sequences, maxlen=max\_length, padding='post', truncating='post')

    return train\_padded, validation\_padded, test\_padded

def create\_tensors\_dataloaders(train\_padded, validation\_padded, test\_padded, train\_list, validation\_list, test\_list, batch\_size=64):

    """Converts padded sequences and labels to PyTorch tensors and DataLoaders."""

    train\_labels\_np = np.array([label for \_, label in train\_list])

    validation\_labels\_np = np.array([label for \_, label in validation\_list])

    test\_labels\_np = np.array([label for \_, label in test\_list])

    train\_data = torch.tensor(train\_padded, dtype=torch.long)

    train\_labels\_tensor = torch.tensor(train\_labels\_np, dtype=torch.float32).view(-1, 1)

    validation\_data = torch.tensor(validation\_padded, dtype=torch.long)

    validation\_labels\_tensor = torch.tensor(validation\_labels\_np, dtype=torch.float32).view(-1, 1)

    test\_data = torch.tensor(test\_padded, dtype=torch.long)

    test\_labels\_tensor = torch.tensor(test\_labels\_np, dtype=torch.float32).view(-1, 1)

    train\_dataset = TensorDataset(train\_data, train\_labels\_tensor)

    validation\_dataset = TensorDataset(validation\_data, validation\_labels\_tensor)

    test\_dataset = TensorDataset(test\_data, test\_labels\_tensor)

    train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

    validation\_loader = DataLoader(validation\_dataset, batch\_size=batch\_size)

    test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size)

    return train\_loader, validation\_loader, test\_loader

def process\_text(train\_list, validation\_list, test\_list, num\_words=20000, max\_length=200, batch\_size=64):

    """Execute the text processing pipeline."""

    train\_sentences = decode\_clean\_text(train\_list)

    validation\_sentences = decode\_clean\_text(validation\_list)

    test\_sentences = decode\_clean\_text(test\_list)

    train\_padded, validation\_padded, test\_padded = tokenize\_pad\_sequences(train\_sentences, validation\_sentences, test\_sentences, num\_words, max\_length)

    train\_loader, validation\_loader, test\_loader = create\_tensors\_dataloaders(train\_padded, validation\_padded, test\_padded, train\_list, validation\_list, test\_list, batch\_size)

    return train\_loader, validation\_loader, test\_loader

train\_loader, validation\_loader, test\_loader = process\_text(train\_list, validation\_list, test\_list)

We start creating a remove\_html\_tags function, this function removes HTML tags from the text. Second, we have the decode\_clean\_text function, this one decodes the strings and then applies the remove\_html\_tags function. Then we used the tokenize\_pad\_sequences function to tokenize the text and after that the pad the sequences up to a maximum length. Next, we transform the sequences and the labels into Pytorch tensors with the create\_tensors\_dataloaders, this function also creates the DataLoader objects. Finally, the process\_text function executes all the functions to obtain the variables that we will use in the model.

* Data Exploration

A graph of a graph

AI-generated content may be incorrect.

A graph with a blue line

AI-generated content may be incorrect.

A graph with a blue line

AI-generated content may be incorrect.

The graphs show two important aspects of the data: the distribution of the review labels and the distribution of the length of the reviews. Then, we show as a bar chart the distribution of the training labels, we can observe a balanced dataset with the same number of positive and negative reviews. This balance is important to train a classification model in the right way, as it prevents the model from being biased towards a single sentiment class. The lengths of the reviews are plotted in two histograms showing the same data. These histograms indicate that most opinions are quite short, and their length is concentrated below 200 words. The distribution is skewed to the right, the number of opinions decreases as the length increases and some few opinions exceed 800 words. Knowing this about the length of reviews is important for making informed decisions during the modeling process, e.g., choosing a reasonable maximum sequence length.



## In addition, here are some other details we found out about the training data:

## We can see that most of the common words are “the”, “and” and ‘of’, but also “movie” and similar words.

## On average, the words are about 4.67 characters long.

## There are 219,700 unique words, so the vocabulary is quite large.

## The length of reviews varies widely (standard deviation of 173.50 words).

## There are 81,191 HTML tags.

## There are 1,066,364 special characters.

## Model Development

* Model Implementation:

## Here is the code we use for implementation of the model:

def create\_rnn\_model(vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate, device):

    """Creates and returns an RNN model with normalization."""

    class RNN(nn.Module):

        def \_\_init\_\_(self, vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate):

            super(RNN, self).\_\_init\_\_()

            self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

            self.embedding\_norm = nn.LayerNorm(embedding\_dim)

            self.gru = nn.GRU(embedding\_dim, hidden\_dim, num\_layers=num\_layers, bidirectional=bidirectional, batch\_first=True, dropout=dropout\_rate if num\_layers > 1 else 0)

            self.gru\_norm = nn.LayerNorm(hidden\_dim \* 2 if bidirectional else hidden\_dim)

            self.dropout = nn.Dropout(dropout\_rate)

            self.fc = nn.Linear(hidden\_dim \* 2 if bidirectional else hidden\_dim, output\_dim)

            self.fc\_norm = nn.LayerNorm(hidden\_dim \* 2 if bidirectional else hidden\_dim)

            self.sigmoid = nn.Sigmoid()

        def forward(self, text):

            embedded = self.embedding(text)

            embedded\_norm = self.embedding\_norm(embedded) # Normalize embedded.

            output, hidden = self.gru(embedded\_norm)

            if self.gru.bidirectional:

                hidden = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)

            else:

                hidden = hidden[-1, :, :]

            hidden\_norm = self.gru\_norm(hidden) # Normalize hidden.

            hidden\_drop = self.dropout(hidden\_norm)

            hidden\_fc\_norm = self.fc\_norm(hidden\_drop) # Normalize linear input.

            out = self.fc(hidden\_fc\_norm)

            return self.sigmoid(out)

    return RNN(vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate).to(device)

# Create the model

model = create\_rnn\_model(vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate, device)

* Model Architecture:

The model we build is a Sequence to Vector model, used mostly when we want to determine is the sentiment is positive or negative, or if we are classifying emails, if the email is spam or not. So, this works as in each time step the model receives an input, process that input and we can ignore the output in that step until the last step when we obtain the result.

A diagram of a software process

AI-generated content may be incorrect.

In the Embedding layer, we convert each word into a number. Then, the model takes these numbers and convert them into vectors, a list of numbers that represent the word. In the next layer, all these vectors are normalized to have the same scale. Then, they go through the GRU layer for processing. The output of this layer is also normalized and then the dropout process is performed to avoid overfitting. Next, we reduce the output of the normalized GRU layer to a single number with the Sigmoid function. Finally, we convert that number to a probability between 0 and 1, to predict whether the sentence is positive or negative. We used the same architecture to configure and test a model with LSTM layer.

## Training and Evaluation

* Experimental Setup:
  + For setup for the experiments, we use Google Colab and when the limitation of this platform did not allow us to run, we run it locally.
  + We use Python 3.12.10
  + We used the next initial parameters:
    - vocab\_size = 20000
    - embedding\_dim = 32
    - hidden\_dim = 64
    - output\_dim = 1
    - num\_layers = 4
    - bidirectional = True
    - dropout\_rate = 0.4
    - weight\_decay = 1e-5
* Training code:

We executed the training process with the train\_validate function, this manages training as well as validation on a specified number of epochs. For each epoch, the model starts for training through model.train(). Next, training data is batched, and the gradients of the optimizer are reset on each batch by using optimizer. The model predicts, and the Binary Cross-Entropy loss is computed by comparing the predicted output with the real labels. The loss is back propagated, and the function update the model parameters. We also applied weight decay and dropout in order to prevent overfitting by not letting huge weights.  
After processing training data, the model executes the model.eval() to compute the performance. Validation loss and accuracy are computed in the same way as training but without backpropagation. We tracked metrics across epochs by graphing it through the plot\_training\_results function.  
  
We add to the training loop an early stopping mechanism. So, if validation loss fails to decrease for several consecutive epochs (we can set this parameter), the training is terminated early. This prevents unnecessary computation on the training set.

def train\_and\_validate(model, train\_loader, validation\_loader, epochs=20, patience=5):

    """Trains and validates the model with early stopping."""

    torch.cuda.empty\_cache()

    criterion = nn.BCELoss()

    optimizer = optim.Adam(model.parameters())

    train\_losses = []

    val\_losses = []

    train\_accuracies = []

    val\_accuracies = []

    best\_val\_loss = float('inf')

    counter = 0

    print("{:<10} {:<15} {:<15} {:<15} {:<15}".format("Epoch", "Train Loss", "Train Acc", "Val Loss", "Val Acc"))

    print("-" \* 65)

    for epoch in range(epochs):

        model.train()

        total\_train\_loss = 0

        correct\_train = 0

        total\_train = 0

        for batch\_text, batch\_labels in train\_loader:

            batch\_text = batch\_text.to(model.embedding.weight.device)

            batch\_labels = batch\_labels.to(model.embedding.weight.device)

            optimizer.zero\_grad()

            predictions = model(batch\_text)

            loss = criterion(predictions, batch\_labels)

            loss.backward()

            optimizer.step()

            total\_train\_loss += loss.item()

            predicted\_train = torch.round(predictions)

            correct\_train += (predicted\_train == batch\_labels).sum().item()

            total\_train += batch\_labels.size(0)

        train\_accuracy = correct\_train / total\_train

        avg\_train\_loss = total\_train\_loss / len(train\_loader)

        train\_losses.append(avg\_train\_loss)

        train\_accuracies.append(train\_accuracy)

        model.eval()

        total\_val\_loss = 0

        correct\_val = 0

        total\_val = 0

        with torch.no\_grad():

            for batch\_text, batch\_labels in validation\_loader:

                batch\_text = batch\_text.to(model.embedding.weight.device)

                batch\_labels = batch\_labels.to(model.embedding.weight.device)

                predictions = model(batch\_text)

                loss = criterion(predictions, batch\_labels)

                total\_val\_loss += loss.item()

                predicted\_val = torch.round(predictions)

                correct\_val += (predicted\_val == batch\_labels).sum().item()

                total\_val += batch\_labels.size(0)

        val\_accuracy = correct\_val / total\_val

        avg\_val\_loss = total\_val\_loss / len(validation\_loader)

        val\_losses.append(avg\_val\_loss)

        val\_accuracies.append(val\_accuracy)

        print("{:<10} {:<15.4f} {:<15.4f} {:<15.4f} {:<15.4f}".format(epoch + 1, avg\_train\_loss, train\_accuracy, avg\_val\_loss, val\_accuracy))

        if avg\_val\_loss < best\_val\_loss:

            best\_val\_loss = avg\_val\_loss

            counter = 0

        else:

            counter += 1

            if counter >= patience:

                print("Early stopping triggered!")

                break

    return train\_losses, val\_losses, train\_accuracies, val\_accuracies

def plot\_training\_results(train\_losses, val\_losses, train\_accuracies, val\_accuracies):

    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)

    plt.plot(train\_losses, label='Train Loss')

    plt.plot(val\_losses, label='Validation Loss')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.legend()

    plt.subplot(1, 2, 2)

    plt.plot(train\_accuracies, label='Train Accuracy')

    plt.plot(val\_accuracies, label='Validation Accuracy')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    plt.legend()

    plt.show()

# Train and validate the model

train\_losses, val\_losses, train\_accuracies, val\_accuracies = train\_and\_validate(model, train\_loader, validation\_loader)

# Plot the results

plot\_training\_results(train\_losses, val\_losses, train\_accuracies, val\_accuracies)

## Results and Analysis

When we start to execute the model in its first steps of development, we found that we are not using normalization, dropout or weight decay. This led us to a bad performance and to overfitting the data. Implementing those parameters, we improved the regularization and prevent the overfitting, promoting a model with a better distribution between the neurons, so, for us was important to understand the implications of configuring a good model.

When we obtained a good model, with a strong code. We add the hyperparameter tuning with Otuna.

from sklearn.base import BaseEstimator, ClassifierMixin

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import accuracy\_score

from sklearn.pipeline import Pipeline

# Convert train\_list and test\_list to numpy arrays.

def convert\_to\_numpy(data\_list):

    """Converts a list of (text, label) tuples to NumPy arrays."""

    texts = [text.decode('utf-8') for text, \_ in data\_list]

    labels = np.array([label for \_, label in data\_list])

    return texts, labels

train\_texts, train\_labels = convert\_to\_numpy(train\_list)

test\_texts, test\_labels = convert\_to\_numpy(test\_list)

# Tokenize and pad the sequences

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

def tokenize\_pad(texts, max\_length=200, num\_words=20000):

    tokenizer = Tokenizer(num\_words=num\_words, oov\_token="<OOV>")

    tokenizer.fit\_on\_texts(texts)

    sequences = tokenizer.texts\_to\_sequences(texts)

    padded\_sequences = pad\_sequences(sequences, maxlen=max\_length, padding='post', truncating='post')

    return padded\_sequences

train\_data = tokenize\_pad(train\_texts)

test\_data = tokenize\_pad(test\_texts)

# device

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

class PyTorchGRUClassifier(BaseEstimator, ClassifierMixin):

    def \_\_init\_\_(self, vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate, weight\_decay, epochs=10, patience=3):

        self.vocab\_size = vocab\_size

        self.embedding\_dim = embedding\_dim

        self.hidden\_dim = hidden\_dim

        self.output\_dim = output\_dim

        self.num\_layers = num\_layers

        self.bidirectional = bidirectional

        self.dropout\_rate = dropout\_rate

        self.weight\_decay = weight\_decay

        self.epochs = epochs

        self.patience = patience

        self.model = None

    def fit(self, X, y):

        print("Model Parameters:")

        print(f"  Vocab Size: {self.vocab\_size}")

        print(f"  Embedding Dim: {self.embedding\_dim}")

        print(f"  Hidden Dim: {self.hidden\_dim}")

        print(f"  Output Dim: {self.output\_dim}")

        print(f"  Num Layers: {self.num\_layers}")

        print(f"  Bidirectional: {self.bidirectional}")

        print(f"  Dropout Rate: {self.dropout\_rate}")

        print(f"  Weight Decay: {self.weight\_decay}")

        print(f"  Epochs: {self.epochs}")

        print("="\*50)

        train\_dataset = TensorDataset(torch.tensor(X, dtype=torch.long), torch.tensor(y, dtype=torch.float32).view(-1, 1))

        train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

        val\_size = int(0.2 \* len(train\_dataset))

        train\_size = len(train\_dataset) - val\_size

        train\_subset, val\_subset = torch.utils.data.random\_split(train\_dataset, [train\_size, val\_size])

        validation\_loader = DataLoader(val\_subset, batch\_size=32)

        class RNN(nn.Module):

            def \_\_init\_\_(self, vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate):

                super(RNN, self).\_\_init\_\_()

                self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

                self.embedding\_norm = nn.LayerNorm(embedding\_dim)

                self.gru = nn.GRU(embedding\_dim, hidden\_dim, num\_layers=num\_layers, bidirectional=bidirectional, batch\_first=True, dropout=dropout\_rate if num\_layers > 1 else 0)

                self.gru\_norm = nn.LayerNorm(hidden\_dim \* 2 if bidirectional else hidden\_dim)

                self.dropout = nn.Dropout(dropout\_rate)

                self.fc = nn.Linear(hidden\_dim \* 2 if bidirectional else hidden\_dim, output\_dim)

                self.fc\_norm = nn.LayerNorm(hidden\_dim \* 2 if bidirectional else hidden\_dim)

                self.sigmoid = nn.Sigmoid()

            def forward(self, text):

                embedded = self.embedding(text)

                embedded\_norm = self.embedding\_norm(embedded)

                output, hidden = self.gru(embedded\_norm)

                if bidirectional:

                    hidden = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)

                else:

                    hidden = hidden[-1, :, :]

                hidden\_norm = self.gru\_norm(hidden)

                hidden\_drop = self.dropout(hidden\_norm)

                hidden\_fc\_norm = self.fc\_norm(hidden\_drop)

                out = self.fc(hidden\_fc\_norm)

                return self.sigmoid(out)

        self.model = RNN(self.vocab\_size, self.embedding\_dim, self.hidden\_dim, self.output\_dim, self.num\_layers, self.bidirectional, self.dropout\_rate).to(device)

        train\_validate(self.model, train\_loader, validation\_loader, self.epochs, self.patience)

        return self

    def predict(self, X):

        test\_dataset = TensorDataset(torch.tensor(X, dtype=torch.long))

        test\_loader = DataLoader(test\_dataset, batch\_size=32)

        self.model.eval()

        predictions = []

        with torch.no\_grad():

            for batch\_text, in test\_loader:

                batch\_text = batch\_text.to(device)

                preds = self.model(batch\_text)

                predictions.extend(torch.round(preds).cpu().numpy())

        return np.array(predictions).flatten()

pipeline = Pipeline([

    ('gru', PyTorchGRUClassifier(vocab\_size=20000, output\_dim=1, embedding\_dim=8, hidden\_dim=16, num\_layers=3, bidirectional=True, dropout\_rate=0.5, weight\_decay=1e-5, epochs=10, patience=3))

])

param\_grid = {

    'gru\_\_embedding\_dim': [16, 32, 64],

    'gru\_\_hidden\_dim': [16, 32, 64],

    'gru\_\_num\_layers': [2, 3],

    'gru\_\_bidirectional': [True, False],

    'gru\_\_dropout\_rate': [0.4, 0.5],

    'gru\_\_weight\_decay': [1e-3, 1e-4],

    'gru\_\_epochs': [10],

    'gru\_\_patience': [3]

}

grid\_search = GridSearchCV(pipeline, param\_grid, cv=3, verbose=1)

grid\_search.fit(train\_data, train\_labels)

if grid\_search.best\_estimator\_:

    print("Best parameters:", grid\_search.best\_params\_)

    print("Best score:", grid\_search.best\_score\_)

    # Evaluate on test data

    predictions = grid\_search.predict(test\_data)

    accuracy = accuracy\_score(test\_labels, predictions)

    print(f"Test Accuracy: {accuracy}")

else:

    print("GridSearchCV did not complete successfully.")

GridSearch executes several tests with different parameters that we already tested in our first executions.

param\_grid = {

    'gru\_\_embedding\_dim': [16, 32, 64],

    'gru\_\_hidden\_dim': [16, 32, 64],

    'gru\_\_num\_layers': [2, 3],

    'gru\_\_bidirectional': [True, False],

    'gru\_\_dropout\_rate': [0.4, 0.5],

    'gru\_\_weight\_decay': [1e-3, 1e-4],

    'gru\_\_epochs': [10],

    'gru\_\_patience': [3]

}

Also, we wanted to print the parameter for each execution and the results. To have a good perspective of what was happening during the executions

    print("Model Parameters:")

        print(f"  Vocab Size: {self.vocab\_size}")

        print(f"  Embedding Dim: {self.embedding\_dim}")

        print(f"  Hidden Dim: {self.hidden\_dim}")

        print(f"  Output Dim: {self.output\_dim}")

        print(f"  Num Layers: {self.num\_layers}")

        print(f"  Bidirectional: {self.bidirectional}")

        print(f"  Dropout Rate: {self.dropout\_rate}")

        print(f"  Weight Decay: {self.weight\_decay}")

        print(f"  Epochs: {self.epochs}")

        print("="\*50)

A screenshot of a computer

AI-generated content may be incorrect.

From these executions we extract the parameters where GRU performs better, and we execute with these parameters LSTM model and we obtain the following:   
  


From the results of both models with the same hyperparameter, and from the experiments we executed, is possible to observe that the model configured with GRU layers was more stable and consistent than the LSTM-enabled model even with identical hyperparameters. This difference can be attributed to the fact that GRU layers are simpler than LSTM layers, as they have fewer gates and therefore require fewer parameters, allowing for faster and more reliable convergence.

We also try hyperparameter tunning with Optuna:

def objective(trial):

    # Hyperparameters to tune

    embedding\_dim = trial.suggest\_int('embedding\_dim', 8, 32)

    hidden\_dim = trial.suggest\_int('hidden\_dim', 8, 32)

    num\_layers = trial.suggest\_int('num\_layers', 2, 4)

    bidirectional = trial.suggest\_categorical('bidirectional', [True, False])

    dropout\_rate = trial.suggest\_float('dropout\_rate', 0.1, 0.5)

    weight\_decay = trial.suggest\_float('weight\_decay', 1e-6, 1e-1, log=True)

    # Model Parameters

    vocab\_size = 20000

    output\_dim = 1

    # Instantiate the model

    model = create\_rnn\_model(vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate, device)

    # Loss function and optimizer

    criterion = nn.BCELoss()

    optimizer = optim.Adam(model.parameters(), weight\_decay=weight\_decay)

    # Training Loop

    epochs = 20

    for epoch in range(epochs):

        model.train()

        for batch\_text, batch\_labels in train\_loader:

            batch\_text = batch\_text.to(model.embedding.weight.device)

            batch\_labels = batch\_labels.to(model.embedding.weight.device)

            optimizer.zero\_grad()

            predictions = model(batch\_text)

            loss = criterion(predictions, batch\_labels)

            loss.backward()

            optimizer.step()

    # Validation

    model.eval()

    correct = 0

    total = 0

    with torch.no\_grad():

        for batch\_text, batch\_labels in validation\_loader:

            batch\_text = batch\_text.to(model.embedding.weight.device)

            batch\_labels = batch\_labels.to(model.embedding.weight.device)

            predictions = model(batch\_text)

            predicted = torch.round(predictions)

            total += batch\_labels.size(0)

            correct += (predicted == batch\_labels).sum().item()

    accuracy = correct / total

    return accuracy

def print\_trial\_results(study):

    print("{:<6} {:<10} {:<15} {:<15} {:<10} {:<15} {:<15} {:<15}".format(

        "Trial", "Accuracy", "Embedding", "Hidden", "Layers", "Bidirect", "Dropout", "Weight Decay"

    ))

    print("-" \* 115)

    for trial in study.trials:

        if trial.state == optuna.trial.TrialState.COMPLETE:

            print("{:<6} {:<10.4f} {:<15} {:<15} {:<10} {:<15} {:<15.6f} {:<15.6e}".format(

                trial.number,

                trial.value,

                trial.params['embedding\_dim'],

                trial.params['hidden\_dim'],

                trial.params['num\_layers'],

                trial.params['bidirectional'],

                trial.params['dropout\_rate'],

                trial.params['weight\_decay']

            ))

# Optuna study

study = optuna.create\_study(direction='maximize')

study.optimize(objective, n\_trials=20, show\_progress\_bar=False)

# Print table

print\_trial\_results(study)

# Best hyperparameters

best\_params = study.best\_params

print("\nBest Hyperparameters:", best\_params)

The results were almost the same, but the difference is the execution time between GridSearch and Optuna. Optuna is faster than GridSearch.

## Improvement or Further Experimentation

Future improvements could include creating an application to connect with the model to create automation to react to the negatives reviews and check a possible action to solve them. In addition, this is not the only application this kind of models could have, we can implement these networks to handle sequential data, like time series, text or audio signals.

Another possible implementation could be using RNN along with CNN to improve the task related to text. This relation could be good to identify more complex patterns or dependencies, for example in natural language applications. Even though CNN are more used for pictures analysis, they can also work with text. When they process text, they use filters that scan for short, nearby sequences of words that often appear together. This helps the model understand small, meaningful parts of the text.

## Conclusions

One important fact is that the dataset has an equal number of classes. This kind of balance helps the model to avoid the bias towards one class, resulting into a better sentiment analysis model.  
  
Another important fact that we considered was the distribution of the review lengths, because in the dataset there are many short reviews mixed with other longer one. We implemented padding techniques to handle these different length sequences to allow the model to handle even short and long reviews.  
  
We conclude that working with both layer option GRU and LSTM at the same time was good, because allow us to learn the difference between each other and to select what is appropriate for this task. GRUs are designed to handle sequential data and are effective at capturing the dependencies between words in a sentence, which is essential for sentiment analysis.  
  
Handlining the overfitting that we experimented at the first executions in the model, was the correct choice. Implementing Dropout and L2 regularization (weight decay) is a standard in any RNN model and helps to preventing overfitting.

## References

*CS 230 - Recurrent Neural Networks Cheatsheet*. (n.d.). https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

*imdb\_reviews*. (n.d.). TensorFlow. https://www.tensorflow.org/datasets/catalog/imdb\_reviews

Kevin, & Kevin. (2024, August 21). Coding a Recurrent Neural Network (RNN) from scratch using Pytorch. *SolarDevs – Amazon Web Services Consulting*. https://solardevs.com/blog/rnn-from-scratch-pytorch/

Luay, M. (2023, September 22). Sentiment Analysis using Recurrent Neural Network(RNN),Long Short Term Memory(LSTM) and Convolutional Neural Network(CNN) with Keras. *Medium*. https://medium.com/@muhammadluay45/sentiment-analysis-using-recurrent-neural-network-rnn-long-short-term-memory-lstm-and-38d6e670173f

Mısra Turp. (2022, March 20). *Fool-proof RNN explanation | What are RNNs, how do they work?* [Video]. YouTube. https://www.youtube.com/watch?v=y9PLF2GsD-c

Singh, R. (2024, November 13). Recurrent Neural Network (RNN) - Rishabh Singh - Medium. *Medium*. https://medium.com/@RobuRishabh/recurrent-neural-network-rnn-8412b9abd755