# Final Project Report

Project Title: Sentiment analysis with RNN.

Course Name: CS 551 – Deep Learning.

Professor's Name: Dr. Alaoui Mhamdi.

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, which in this case is 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. The goal of this exercise was not only to understand the practical applications of these models but also to develop transferable skills that could be applied in 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 sub-field of Natural Language Processing (NLP), which consists of 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. Key components of the project include data preprocessing, careful model architecture design (with both LSTM and GRU variants explored), hyperparameter optimization, and thorough model evaluation.

The challenge of this project is to use RNNs effectively 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. Moreover, ensuring reproducibility is critical, as variations in data partitioning and initialization parameters can lead to inconsistent results.

Sentiment analysis provides valuable insights to companies, researchers and individuals by unveiling hidden patterns in text data. Given its wide-ranging applications across various industries, mastering RNN models becomes essential to harness these insights effectively, which is precisely what our project aims to achieve.

Building on this premise, the project focuses on several principal points: first, developing a clear and reproducible workflow for conducting sentiment analysis using RNNs; secondly, implementing and comparing the performance of LSTM and GRU-based RNN models on the IMDB dataset; third, optimizing hyperparameters while rigorously evaluating model performance, using appropriate metrics; and finally, documenting the entire process in depth, from data preprocessing to model evaluation, to guarantee reproducibility.

## Dataset Selection and Preprocessing steps

* **Dataset: IMDB review dataset**

The selected dataset for this project is the IMDB dataset, a text-based binary sentiment classification collection comprising 25,000 highly polar movie reviews for training, and an additional 25,000 reviews for testing. We obtained this dataset from TensorFlow Datasets and choose it because it is a popular benchmark for classification tasks. The dataset presented us with a good balance between positive and negative reviews, which made it ideal for training and evaluating a sentiment analysis model, while its simplicity offered an excellent learning opportunity for applying RNNs in practical applications.

* **Loading the data**

We used the following TensorFlow function to load the data and then split it into three parts, namely 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**

Then, we implemented 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)

As seen above, we started by creating a *remove\_html\_tags* function that removes any HTML tags from the text. Next, we defined the *decode\_clean\_text* function, which decodes the strings and applies *remove\_html\_tags* to clean them. After that, we used the *tokenize\_pad\_sequences* function to tokenize the text and pad the resulting sequences to a fixed maximum length. Then, we used the *create\_tensors\_dataloaders* function to transform the sequences and labels into Pytorch tensors to create the corresponding DataLoader objects. Finally, the *process\_text* function executed all of these steps to obtain the variables required for our 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 above showed two important aspects of the dataset: the distribution of review labels and the distribution of reviews lengths. First, a bar chart showing the distribution of training labels revealed that the dataset is balanced, with an equal number of positive and negative reviews. This balance is crucial for training a classification model effectively, as it prevents bias toward a single sentiment class.

Next, two histograms depicted the review lengths. They indicated that most reviews are relatively short, with the majority containing fewer than 200 words. The distribution is right-skewed, meaning that the number of reviews decreases as the length increases, and only a few reviews exceed 800 words. This insight was vital for making informed decisions during the modeling process, such as choosing an appropriate maximum sequence length.



## Finally, as shown in the above histogram, we uncovered several additional characteristics of the training data:

## **Common words:** The most common words in the dataset are “the”, “and” and ‘of’, along with domain-specific words like “movie”.

## **Average Word Length:** On average, the words are about 4.67 characters long.

## **Vocabulary size:** There are 219,700 unique words in the dataset, indicating a substantial vocabulary.

## **Review Length Variability:** The lengths of reviews varies widely, with a standard deviation of 173.50 words.

## **HTML Tags:** We found 81,191 HTML tags embedded with the reviews.

## **Special Characters:** In total, there are 1,066,364 special characters present.

## Model Development

* **Model Implementation**

## The model was implemented as follows:

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)

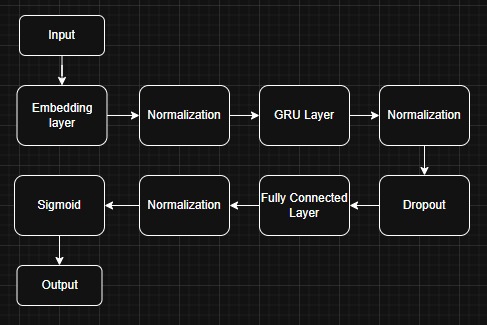
    return RNN(vocab\_size, embedding\_dim, hidden\_dim, output\_dim, num\_layers, bidirectional, dropout\_rate).to(device)

# 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 built is a Sequence-to-Vector model, typically used to determine if the sentiment is positive or negative, or to classify emails as spam or not. The way it works is that at each time step the model receives an input and processes it, while the intermediate outputs are disregarded until the final step produces the overall result.



In the Embedding layer, we first converted each word into a number. Then, the model transformed the numbers into vectors that captured the semantic properties of the words. Next, we normalized the vectors to ensure a uniform scale before feeding them into the GRU layer for further processing. The output of the GRU layer was also normalized and dropout was applied to avoid overfitting. After that, we reduced the output of the normalized GRU layer to a single number with the Sigmoid function. Finally, we converted that number to a probability between 0 and 1 to predict whether the sentence was positive or negative. The same architecture was used to configure and test a model with an LSTM layer. This was done to allow us to compare both architectures against the IMDB dataset and choose the one that would provide the best result.

## **Training and Evaluation**

* **Experimental Setup**
  + To setup the experiments, we used Google Colab; however, when the platform’s limitations prevented us from running our code, we executed the experiments locally.
  + We used Python 3.12.10
  + The initial parameters were set as follows:
    - 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, which managed both training and validation over a specified number of epochs. For each epoch, the model was set to training mode using *model.train()*. Next, training data was batched, and the optimizer’s gradients were reset for each batch. The model then made predictions, and the Binary Cross-Entropy loss was computed by comparing the predicted outputs with the true labels. The loss was backpropagated, and the function updated the model parameters accordingly. We also applied weight decay and dropout to help prevent overfitting by limiting the magnitude od the weights.  
After processing the training data, the model was switched to evaluation mode using *model.eval()* to compute the performance metrics. The validation loss and accuracy were computed similarly to the training phase but without backpropagation. We tracked metrics across epochs by graphing them with the *plot\_training\_results* function.  
  
Additionally, we integrated an early stopping mechanism into the training loop. If the validation loss failed to decrease for several consecutive epochs (a parameter we could set), the training was terminated early to prevent 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**

During the early stages of model development, we noticed that we were not applying normalization, dropout, or weight decay. This oversight led to poor performance and significant overfitting. By implementing these regularization techniques, we improved the model’s performance, prevented overfitting, and promoted a more balanced distribution of activations among neurons. This process was important for us, and allowed us to demonstrate the importance of configuring good model parameters.

Once we were dealing with a good model, with robust code, we enhanced it even further by integrating 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.")

We used GridSearch to execute several tests with different parameter combinations that we had already explored 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]

}

We also printed the parameters for each execution along with the results, to have a clear perspective on the experiment’s progress.

    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.

Based on these experiments, we determined the parameter combinations where the GRU performed best. We then executed the LSTM model with these parameters and observed the following results:   
  


Based on the results of both models using the same hyperparameters and our experiments, we observed that the GRU-based model was more stable and consistent than the LSTM-based one. 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 applied hyperparameter tuning with Optuna to the LSTM model:

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)

While the results were almost the same, the difference is the execution time between GridSearch and Optuna was evident. Optuna was faster than GridSearch.

## **Improvement or Further Experimentation**

Future improvements could include developing an application to connect with the model to automate responses to the negatives reviews and propose potential solutions. these models have broader applications, as they can be implemented to handle various types of sequential data, such as time series, text or audio signals.

Another possible experiment could be combining RNN with CNN to enhance text-processing tasks. This approach could identify more complex patterns or dependencies, for example, in natural language applications. Although CNN is typically used for image analysis, it can also be effective in text processing by using filters to detect short, contiguous sequences of words that frequently occur together. This helps the model identify small, meaningful parts of the text.

## **Conclusions**

One important observation is that the dataset has an equal number of classes. This kind of balance helps the model avoid bias towards one class, resulting into a better sentiment analysis model.  
  
Another important consideration was the distribution of the review lengths. The dataset includes many short reviews mixed with longer ones, so we implemented padding techniques to accommodate these differences and ensure the model could effectively handle all sequence lengths.  
  
We concluded that experimenting with both GRU and LSTM layers was beneficial, as it allowed us to compare their performance and select the most appropriate architecture for the task. In particular, we found that GRU was especially suited to the task at hand, since it is designed to handle sequential data and and effectively capture dependencies between words in a sentence, which is essential for sentiment analysis.  
  
Finally, addressing the overfitting observed during the initial experiments proved essential. Implementing Dropout and L2 regularization (weight decay) is standard practice in RNN models and helps prevent overfitting effectively.

It is worth noting that part of the success of this project was driven by supplementary research and experimentation on additional datasets. Experiments on an EEG time series dataset (EEG Motor Movement/Imagery Dataset) from PhysioNet and a literary text corpus (Alice’s Adventures in Wonderland) from Project Gutenberg provided valuable insights into handling sequential data and optimizing preprocessing strategies. These cross-domain explorations reinforced our methodology and contributed to the robust performance of our sentiment analysis model.

## **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