

Towards Enhanced Text Classification Analysis: Exploring the Advantages of Bi-LSTM over CNN, RNN, and LSTM Models

by

Yinuo Zhang

Thesis Advisor: Prof. Xu Yang

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science
in
Applied Mathematics



Department of Mathematics
University of California, Santa Barbara
June 2025

Abstract

In the pursuit of effectively classifying customer complaints—a critical aspect of customer relationship management—this project develops a robust workflow for automating text classification using Bi-LSTM. Motivated by the limitations of traditional SQL-based data processing and the inadequacies of earlier machine learning methods in handling textual data, this work explores the advantages of deep learning approaches. Among these, Bi-LSTM has been identified as a promising method due to its bidirectional context-capturing capabilities.

The project utilizes a dataset of customer complaints to evaluate the effectiveness of Bi-LSTM, comparing its performance with traditional machine learning models and other neural network architectures such as CNNs and standard LSTMs. Preliminary experiments demonstrate the superiority of Bi-LSTM in accurately capturing the sequential and contextual nature of complaint text, leading to improved classification accuracy.

Despite the promising results, this study acknowledges challenges such as data imbalance and the need for extensive preprocessing. This work provides a comprehensive workflow for implementing Bi-LSTM in text classification tasks and highlights the shortcomings of alternative methods, offering insights into optimal strategies for improving complaint resolution systems.

1 Introduction

During my internship, I encountered a significant challenge in handling customer complaint classification. The company relied heavily on SQL-based data processing, which, while effective for structured numerical data, struggled with the complexities of unstructured text. Customer complaints contained nuanced language, varied sentence structures, and domain-specific terms, making it difficult to categorize them accurately using rule-based SQL queries. Keyword matching and basic filtering led to misclassification, failing to capture the actual sentiment and intent behind customer issues. Recognizing these limitations, I sought a more advanced approach to automate and enhance text classification accuracy. Traditional machine learning approaches, such as Naïve Bayes and Support Vector Machines (SVMs), have improved text classification but remain limited in their ability to model sequential patterns or capture contextual dependencies within text (Minaee et al., 2021). The emergence of deep learning has revolutionized text classification tasks. Convolutional Neural Networks (CNNs) have been applied successfully to capture local text patterns (Kim, 2014), while Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) networks, are well suited for handling sequential data by addressing long-term dependency issues (Hochreiter & Schmidhuber, 1997). However, these models are still constrained by their unidirectional nature, as they process text in only one temporal direction. Bidirectional LSTMs (Bi-LSTMs), first proposed by Schuster & Paliwal (1997), overcome this limitation by analyzing input sequences in both forward and backward directions. This allows Bi-LSTM networks to extract richer context and achieve superior performance in tasks requiring deeper textual understanding (Zhou et al., 2016). Given its effectiveness in learning sequential dependencies and capturing contextual meaning, Bi-LSTM became the primary choice for improving customer complaint classification. In this study, I focus on using Bi-LSTM for classifying customer complaints, comparing its performance against traditional machine learning methods and other deep learning models, such as CNNs and standard LSTMs. By leveraging the sequential and contextual modeling capabilities of Bi-LSTM, I aim to address the limitations of previous approaches and provide a robust framework for automating customer complaint classification, inspired by successful applications of Bi-LSTM in text analysis (Zhang et al., 2018). This work also highlights challenges associated with imbalanced datasets and computational complexity, providing a practical workflow for businesses to enhance their customer feedback systems.

2 Method: Comparative Analysis of Deep Learning Models for Text Classification

In this section, I will compare methods used to classify customer complaints using deep learning models. Each of these steps has the corresponding Python code available on [GitHub](#).

2.1 CNN Structure

Convolutional Neural Networks (CNNs) are a type of deep learning model designed for processing structured grid-like data, such as images and text. Unlike traditional neural networks, CNNs use convolutional layers to extract local patterns and features from input data. Each convolutional layer applies filters that scan small regions of the input, detecting patterns such as edges and textures in images or key word sequences and n-gram features in text classification tasks.

In text classification, CNNs are effective in capturing local dependencies between words by applying convolutional filters over word embeddings. This allows the model to detect meaningful phrases and important contextual patterns. Max pooling layers help to downsample the feature maps by selecting the most relevant features, reducing computational complexity while retaining critical information. As data passes through multiple convolutional and pooling layers, CNNs build a hierarchical representation, capturing both low-level and high-level features. The extracted features are then processed by fully connected layers to perform classification. CNNs have been widely used in sentiment analysis, topic categorization, and document classification due to their efficiency in processing short to medium-length text.

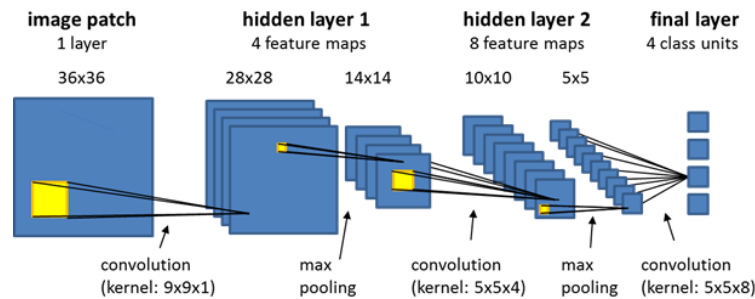


Figure 1: The diagram illustrates the key layers of a Convolutional Neural Network (CNN) and how input features are processed. The input image passes through successive convolutional layers where filters (kernels) scan for local patterns such as edges, textures, and shapes. Max pooling layers follow each convolutional operation, reducing the spatial dimensions by selecting the maximum value within a region, which helps to downsample the feature maps and reduce computational complexity while retaining important features. Multiple layers of convolution and pooling enable hierarchical feature extraction, progressing from low-level details to higher-level representations, eventually leading to fully connected or output layers for classification or prediction tasks. Image taken from [Iza+23].

2.2 RNN Structure

Recurrent Neural Networks (RNNs) are a type of neural network specifically designed for sequential data processing. Unlike traditional feedforward networks, RNNs incorporate a recurrent connection that allows information to persist across time steps. At each time step, the network takes an input and updates its hidden state, which carries information from previous steps. This enables RNNs to capture temporal dependencies and sequential patterns in data, making them useful for tasks such as speech recognition, language modeling, and text classification.

In text classification, RNNs process word sequences one step at a time, using their hidden state to retain context from previous words. The weight matrices control different operations within the network: one connects the input to the hidden state, another manages the recurrent connection that passes information through time, and the final one maps the hidden state to the output. While RNNs perform well for short sequences, they suffer from the vanishing gradient problem, which makes learning long-term dependencies difficult. This limitation is addressed by advanced architectures such as Long Short-Term Memory (LSTM) and Bidirectional LSTM (Bi-LSTM), which improve the ability to retain information over extended sequences.

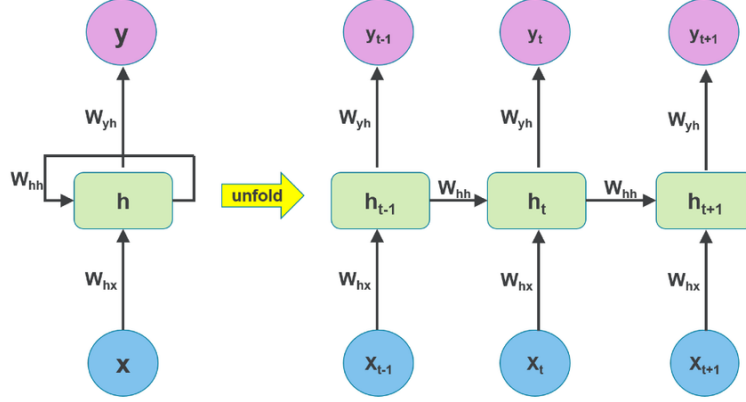


Figure 2: RNN structure

The diagram illustrates an unfolded Recurrent Neural Network (RNN) across multiple time steps $t-1$, and $t+1$. The input at each time step x_t is processed to produce a hidden state h_t , which stores sequential information and is influenced by both the current input and the previous hidden state h_{t-1} . The weight matrices W_{hx} , W_{hh} , and W_{yh} control different operations within the network: W_{hx} connects the input x_t to the hidden state, W_{hh} handles the recurrent connection passing information from the previous hidden state to the next, and W_{yh} maps the hidden state h_t to the output y_t . This structure enables RNNs to model temporal dependencies in sequential data. Image taken from [Ven19].

2.3 LSTM Structure

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed to address the vanishing gradient problem, which makes standard RNNs struggle with learning long-range dependencies. Unlike traditional RNNs, LSTMs introduce memory cells that can store and selectively retain information over long sequences. The architecture of an LSTM cell consists of three key gates: the forget gate, which determines how much past information should be discarded; the input gate, which updates the cell state with new information; and the output gate, which controls what part of the memory is used to generate the current hidden state. These mechanisms, regulated by sigmoid and tanh activation functions, enable LSTMs to capture both short-term and long-term dependencies in sequential data.

LSTMs have been widely used in various applications, including text classification, speech recognition, and machine translation, where context over long sequences is critical. In the context of customer complaint classification, LSTMs offer a significant advantage over traditional machine learning models by understanding word sequences and capturing contextual meaning. However, one limitation of standard LSTMs is their unidirectional processing, meaning they can only utilize past information when making predictions. This limitation reduces their effectiveness in tasks that require understanding both past and future context, a challenge that Bidirectional LSTM (Bi-LSTM) overcomes, which will be discussed in the next section.

2.4 Bi-LSTM Structure

Bidirectional Long Short-Term Memory (Bi-LSTM) is an advanced type of Recurrent Neural Network (RNN) designed to process sequential data more effectively by capturing dependencies from both past and future contexts. Unlike standard LSTM, which processes input only in a forward direction, Bi-LSTM consists of two parallel LSTM layers—one moving forward and another moving backward—allowing the model to leverage information from both directions. This bidirectional capability makes Bi-LSTM particularly effective in natural language processing (NLP) tasks, where context plays a crucial role in determining meaning.

2.4.1 Architecture of Bi-LSTM

A Bi-LSTM consists of two separate LSTM networks that process the input sequence independently:

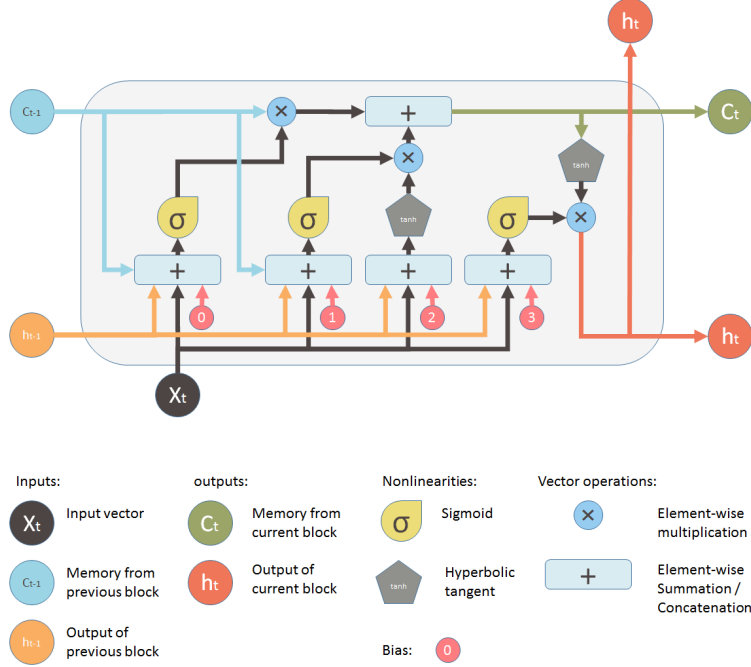


Figure 3: LSTM Structure

The internal structure of an LSTM cell, illustrating how the forget gate f_t , input gate, and output gate work together to manage information flow. The forget gate decides what part of the previous cell state C_{t-1} should be retained, while the input gate updates the cell state C_t using the current input X_t and candidate state. The updated cell state passes through the output gate, which controls what information becomes the current output h_t . Nonlinear activation functions—sigmoid and tanh—ensure smooth and controlled updates, enabling the LSTM to capture both short-term and long-term dependencies. Image taken from [Yan16].

Forward LSTM: Processes the sequence from the first token to the last, capturing dependencies from the past. Backward LSTM: Processes the sequence in reverse, capturing dependencies from the future. Concatenation Layer: Combines the outputs from both directions at each time step. Fully Connected Layers: Maps the extracted features to the output classes. At each time step t , the forward and backward hidden states are computed as follows: At each time step t , the forward and backward hidden states are computed as follows:

$$\vec{h}_t = LSTM_f(x_t, \vec{h}_{t-1}) \quad (1)$$

$$\overleftarrow{h}_t = LSTM_b(x_t, \overleftarrow{h}_{t+1}) \quad (2)$$

$$h_t = [\vec{h}_t; \overleftarrow{h}_t] \quad (3)$$

where h_t represents the final hidden state at time step t , combining both past and future information.

2.4.2 Advantages of Bi-LSTM

Bi-LSTM has several key advantages over standard LSTM: 1.Captures full context: Unlike unidirectional LSTM, which only considers past information, Bi-LSTM also takes future context into account, making it more effective for text understanding. 2.Better performance in NLP tasks: Since meaning in language often depends on both prior and upcoming words, Bi-LSTM improves accuracy in tasks such as sentiment

analysis, named entity recognition, and text classification. 3. More robust learning: By processing data in two directions, Bi-LSTM reduces errors that may arise from limited context availability.

2.4.3 Application in Text Classification

In customer complaint classification, Bi-LSTM helps in capturing long-range dependencies and understanding the relationship between words more effectively. Traditional machine learning models, such as Naïve Bayes or SVM, rely on statistical methods that fail to grasp sequential meaning, while CNNs are effective at extracting local features but lack sequence awareness. Bi-LSTM, on the other hand, excels in modeling sequential patterns and improving classification accuracy.

During training, the model uses embeddings such as Word2Vec, GloVe, or FastText to convert words into dense vector representations. The Bi-LSTM layers then process these embeddings to extract features that reflect both previous and upcoming context. The final dense layers map the learned features to class probabilities using a softmax activation function.

2.4.4 Comparison with Other Models

Model	Processing Direction	Handles Long-Term Dependencies	Captures Full Context	Performance in Text Classification
RNN	Left to Right	No (Vanishing Gradient Issue)	No	Moderate
LSTM	Left to Right	Yes	No	Good
CNN	None (Spatial)	No	No	Good for Local Features
Bi-LSTM	Both Left & Right	Yes	Yes	Best for Sequential Data

Table 1: Comparison of Different Neural Network Models for Text Classification

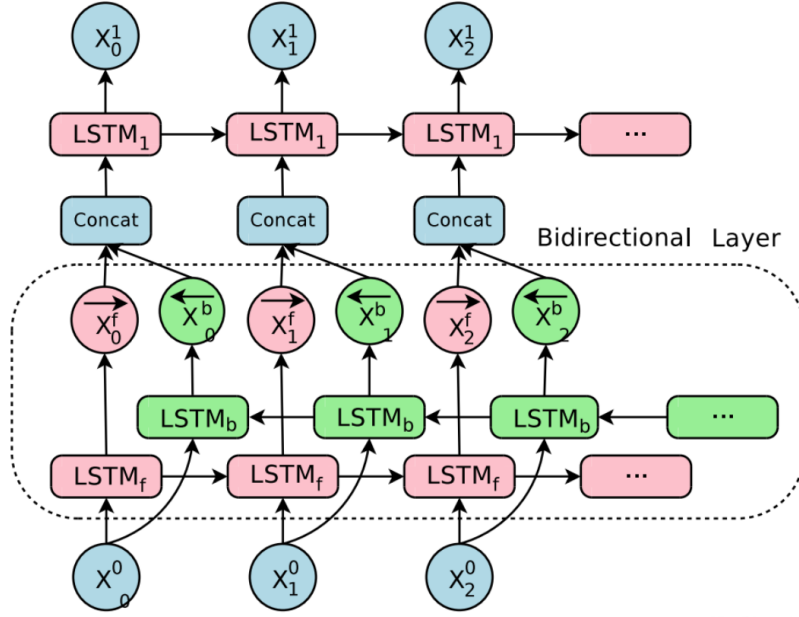


Figure 4: Bi-LSTM Structure. The structure of bi-directional connections in the first layer of the encoder. LSTM layer $LSTM_f$ processes information from left to right, while LSTM layer $LSTM_b$ processes information from right to left. Output from $LSTM_f$ and $LSTM_b$ are first concatenated and then fed to the next LSTM layer $LSTM_1$. The Bi-LSTM structure processes sequences in both forward and backward directions, allowing the model to capture contextual information from both the past and future. This enhances its ability to understand sequential dependencies compared to standard LSTMs, which only process in one direction. Image taken from [WSC+16].

2.5 Data Collection

This study utilizes a dataset of customer complaints related to various financial products, sourced from publicly available records and adapted for classification tasks. Due to privacy considerations, the dataset consists of synthetic data, structured to maintain the statistical characteristics of real-world complaints while ensuring that no personally identifiable information is present. The data set follows the structure of the Kaggle multiclass customer complaint classification dataset (Vikram, 2022) and has been used in similar research studies on text classification[Vik22].

2.6 Data Cleaning

Before training the classification models, it is essential to clean the dataset to ensure consistency, remove irrelevant information, and standardize the textual data. This section describes the data cleaning steps applied to the dataset, supported by visualizations that highlight key characteristics and issues.

2.6.1 Handling Missing Values

The dataset initially contained 162,421 entries with two columns: complaint text and product category. During the initial analysis, a small number of missing values (10 instances) were detected in the complaint text column. Since textual data is the primary input for the classification models, any missing entries were removed to avoid incomplete records in training. After removing these instances, the dataset was reduced to 162,411 entries, ensuring that all remaining records contained meaningful complaint text.

2.6.2 Standardizing Text Format

To prepare the dataset for tokenization and embedding, the complaint text was converted to string format, ensuring uniform data representation. This step is crucial as raw text may sometimes contain numerical values or unrecognized characters that need to be processed consistently.

2.6.3 Dataset Imbalance Analysis

An analysis of the product category distribution revealed significant class imbalance, as shown in the bar chart below. The majority of complaints were related to credit reporting (91,172 complaints), while other categories, such as retail banking and credit card issues, had significantly fewer complaints.

Bar Chart Showing Product Category Distribution:

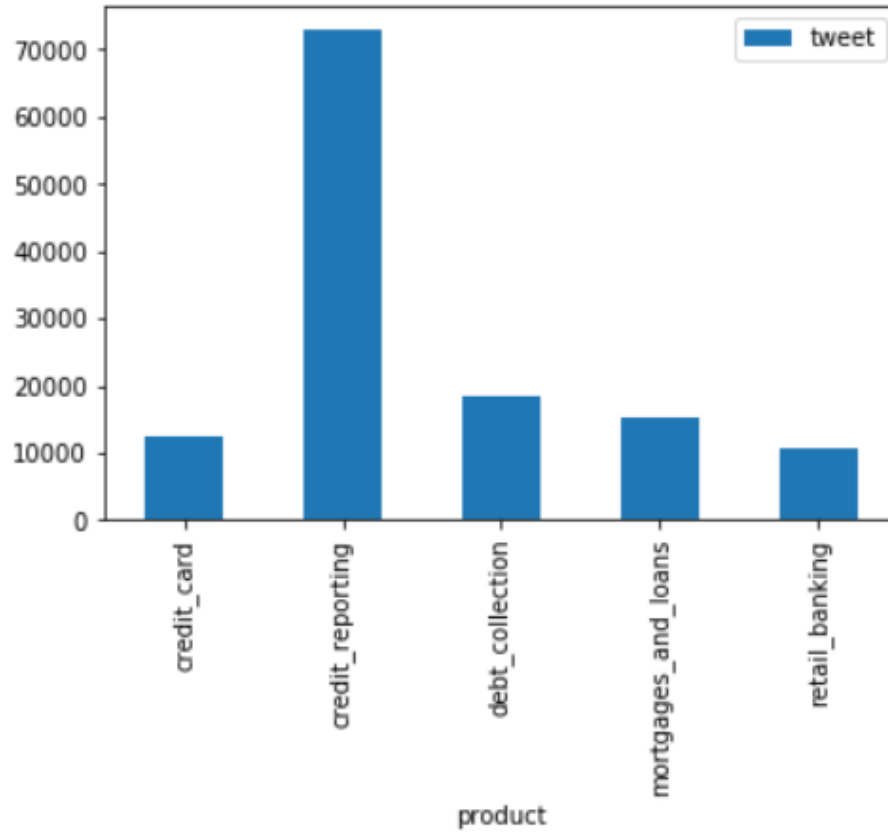


Figure 5: Product Value Counts. This bar chart visualizes the distribution of customer complaints across different product categories, highlighting the class imbalance within the dataset. The majority of complaints pertain to credit reporting, with over 91,172 entries, making it the most frequently reported issue. Other categories, including debt collection, mortgages and loans, credit card issues, and retail banking, exhibit significantly lower counts.

Class imbalance can lead to biased model predictions, where the model favors the dominant category while underperforming on minority classes. To mitigate this issue, techniques such as oversampling, undersampling, or class weighting will be explored in the model training phase.

2.6.4 Word Length Analysis

A word-length analysis was conducted to assess the variability in complaint text length. The results, visualized in the word-length distribution plot, show that complaint texts vary significantly in length, with some

containing only a few words while others are lengthy descriptions of issues.

Word-length distribution plot:

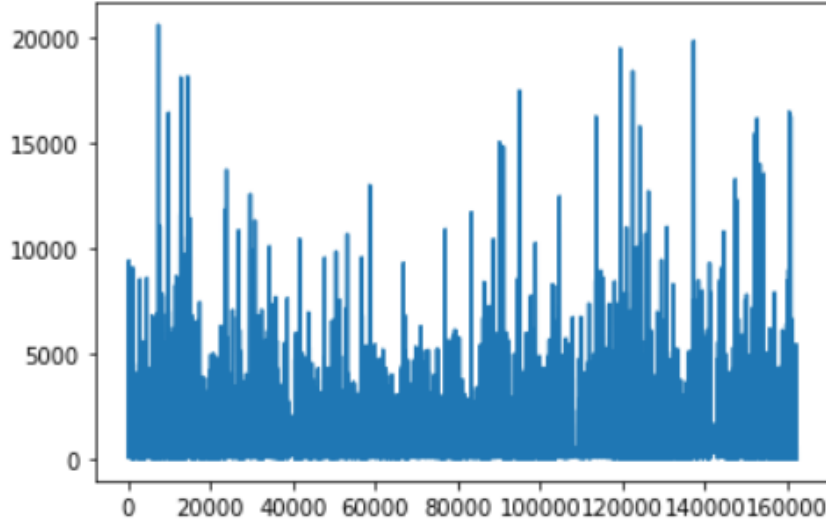


Figure 6: Word Length Value Counts. This plot represents the distribution of word lengths in the complaint dataset, illustrating variations in the number of words per complaint. The x-axis corresponds to individual complaint entries, while the y-axis represents the word count for each complaint. The significant fluctuations in word length suggest a diverse range of complaint descriptions, from short, concise statements to longer, more detailed narratives. Understanding this distribution is crucial for preprocessing steps such as text truncation, padding, and tokenization, which ensure that input sequences are standardized for deep learning models.

Understanding word length distribution helps in setting an appropriate sequence length for tokenization and padding, ensuring that the model captures meaningful information without unnecessary truncation or excessive padding.

2.7 Data Preprocessing

After data cleaning, further preprocessing steps are applied to prepare the dataset for training. This includes splitting the dataset into training and test sets, handling class imbalance, and transforming text data. These steps ensure that the model is trained on a well-balanced and appropriately formatted dataset.

2.7.1 Train-Test Splitting

To evaluate the model's generalization ability, the dataset is split into training and test sets. The training set is used to train the model, while the test set serves as unseen data to assess performance. In this study, 80% of the data is allocated for training, and 20% is reserved for testing.

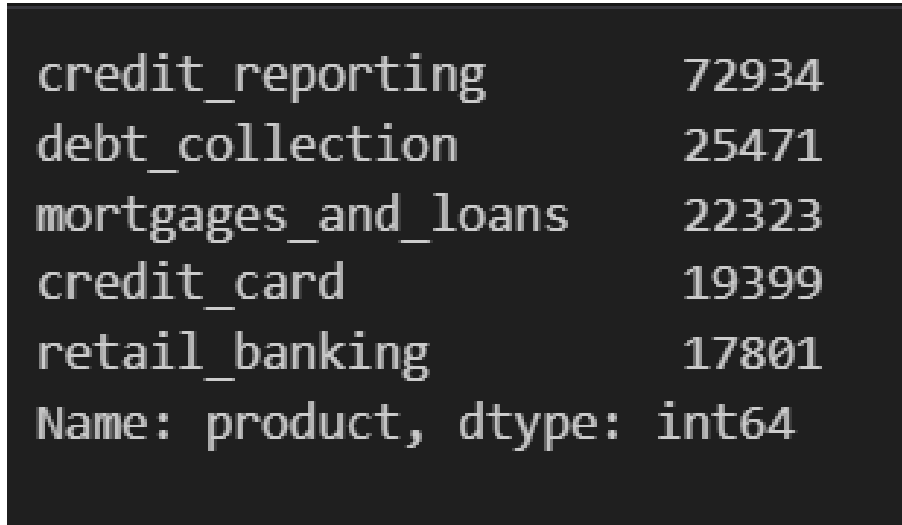
Dataset Split Statistics: Training Set Size: 129,928 records Test Set Size: 32,483 records This ensures that the model is trained on a large portion of the dataset while preserving a significant amount for evaluation.

2.7.2 Handling Class Imbalance with Random Oversampling

As observed in the data cleaning stage, the dataset exhibits a class imbalance, with certain complaint categories appearing significantly more frequently than others. This imbalance can cause the model to favor the dominant classes while underperforming on the minority classes.

To mitigate this issue, random oversampling is applied. This involves randomly duplicating samples from underrepresented classes to balance the dataset. In this study, 7,000 samples were randomly selected from the minority classes (credit card, debt collection, mortgages and loans, and retail banking) and added to the training dataset.

Before Oversampling: Minority class counts were significantly lower than the majority class. After Oversampling: The number of complaints in each category was adjusted, improving balance. Updated Class Distribution after Oversampling:



credit_reporting	72934
debt_collection	25471
mortgages_and_loans	22323
credit_card	19399
retail_banking	17801
Name: product, dtype: int64	

Figure 7: Training Set Counts After Random Oversampling. This table displays the distribution of complaint categories in the training dataset after applying random oversampling to address class imbalance. Initially, the dataset was highly imbalanced, with "credit reporting" significantly overrepresented compared to other categories. To mitigate this issue, random oversampling was performed on the underrepresented classes, increasing their sample sizes to improve model performance and prevent bias toward the majority class. The final counts show a more balanced distribution, allowing the model to learn equally from all categories and improving the generalization ability of the classifier.

2.7.3 Text Preprocessing

Since textual data serves as the input for the classification model, it must be preprocessed into a normalized and structured format. The following transformations are applied:

Stopword Removal: Common English stopwords (e.g., "the," "is," "and") are removed as they do not contribute significantly to classification. Punctuation Removal: Special characters are removed to eliminate noise. Lowercasing: All text is converted to lowercase to ensure consistency. Whitespace Normalization: Extra spaces between words are reduced to maintain text uniformity.

2.7.4 Label Encoding

To convert categorical labels into numerical representations, LabelEncoder from sklearn.preprocessing was used. The training and test labels were first transformed into numerical indices. Since PyTorch requires tensor-based processing, these numerical labels were further converted into PyTorch tensors using torch.tensor(). Additionally, one-hot encoding was applied using torch.nn.functional.one_hot() to ensure the labels were in a format suitable for multi-class classification tasks.

2.7.5 Tokenization and Sequence Padding

For tokenization, `torchtext.data.utils.get_tokenizer()` was utilized to tokenize input text into sequences. A vocabulary was built using `torchtext.vocab.build_vocab_from_iterator()`, limiting the number of unique tokens to a predefined `max_words`. The text sequences were then mapped to their respective token indices within the vocabulary.

To standardize sequence lengths, padding was applied using `torch.nn.utils.rnn.pad_sequence()`, ensuring that all input sequences had the same `max_sequence_length`. This padding was implemented using a value of 0, aligning the data structure for batch processing.

By employing PyTorch-based methods, the data was efficiently preprocessed for training within a deep learning pipeline while maintaining compatibility with Bi-LSTM models.

3 Bi-LSTM Model

In this section, we introduce the architecture of the Bidirectional Long Short-Term Memory (Bi-LSTM) model employed for the task of customer complaint classification. Bi-LSTM extends the standard LSTM by incorporating both forward and backward information, allowing it to capture long-range dependencies from both past and future contexts within textual data. This makes it particularly useful for text classification tasks where understanding the sequential order and relationships between words is crucial.

3.1 Model Architecture

The Bi-LSTM model was implemented using PyTorch, using its flexible deep learning framework. The architecture consists of several key components:

Embedding Layer: An `nn.Embedding` layer was utilized to convert input words into dense vector representations. The embedding dimension was set to 32, ensuring a compact but meaningful representation of words in the dataset.

Bidirectional LSTM Layer: The core of the model is a bidirectional LSTM (`nn.LSTM`), which consists of 128 hidden units and operates in both forward and backward directions. The `batch_first=True` argument ensures that the input tensor dimensions are arranged correctly for PyTorch. **Dropout Layer:** To avoid overfitting, a dropout layer (`nn.Dropout`) with a dropout rate of 0.5 was incorporated.

Fully Connected Layer: The output of the LSTM layer is passed through a linear (`nn.Linear`) layer that maps the hidden representations to the final output classes. Since Bi-LSTM processes text in both directions, the hidden dimension of the fully connected layer is $128 * 2$ to account for both directions. The forward method of the model follows this sequence:

- 1.The input text is passed through the embedding layer to obtain dense vector representations.
- 2.The embeddings are then fed into the Bi-LSTM layer, which processes the input sequence and generates hidden states.
- 3.The last hidden state is selected to capture the overall meaning of the input text.
- 4.Dropout is applied to regularize the network.
- 5.Finally, the processed hidden state is passed through a fully connected layer to generate class predictions.

3.2 Model Initialization and Training Setup

In this section, the Bi-LSTM model is instantiated with the defined parameters and prepared for training. The model utilizes PyTorch for deep learning operations, ensuring computational efficiency and dynamic device assignment based on availability.

3.2.1 Device Configuration and Class Weights

To enhance computational efficiency, the script first checks the availability of GPUs using `torch.cuda.is_available()`. If a GPU is available, computations are performed on CUDA; otherwise, the model runs on the CPU.

To handle class imbalance in the dataset, class weights are assigned on the basis of class distributions. These weights help the model learn equally from all categories instead of being biased towards more frequent

ones. The `torch.tensor()` function is used to define these weights, which are then incorporated into the `CrossEntropyLoss` function.

3.2.2 Loss Function and Optimization Strategy

The loss function is selected based on the task requirements:

`CrossEntropyLoss` is used for multi-class classification, incorporating class weights to address imbalanced data.

Binary Cross Entropy with Logits (`BCEWithLogitsLoss`) is applied if the task is reformulated as multiple independent binary classification problems.

To optimize the training process, `RMSprop` is chosen as the optimizer, with a learning rate of 0.001. This optimizer is well-suited for handling text data as it adaptively adjusts learning rates based on recent gradients.

3.2.3 Evaluation Metrics

To evaluate model performance, an accuracy metric is implemented: The sigmoid activation function is applied to generate probabilities.

Predicted probabilities are rounded to obtain discrete class labels.

The proportion of correctly predicted samples over the total dataset size is computed.

Algorithm 1 Model Optimization and Accuracy Calculation

Require: M : Bi-LSTM Model, η : Learning Rate, W : Model Parameters, Y_{true} : Ground Truth Labels, Y_{pred} : Model Predictions

Ensure: Initialized optimizer and computed accuracy

Step 1: Initialize Optimizer

$\text{optimizer} \leftarrow \text{RMSprop}(M(W), \eta = 0.001)$

Step 2: Compute Predictions

$Y_{\text{pred}} \leftarrow \sigma(Y_{\text{pred}})$

▷ Apply sigmoid activation

$Y_{\text{binary}} \leftarrow \text{round}(Y_{\text{pred}})$

▷ Convert probabilities to binary classification

Step 3: Compute Accuracy

$\text{correct} \leftarrow (Y_{\text{binary}} == Y_{\text{true}}).\text{float}()$

▷ Compare with true labels

$\text{accuracy} \leftarrow \frac{\sum \text{correct}}{\text{len}(Y_{\text{true}})}$

Step 4: Return Accuracy

return accuracy

3.3 Model Training and Evaluation

This section details the training and evaluation process for the Bi-LSTM model implemented using PyTorch. The process includes data preparation, model training, and performance evaluation.

3.3.1 Data Preparation and Model Initialization

Before training, the data is prepared by converting text sequences into numerical representations and padding them to ensure uniform input length. The PyTorch `DataLoader` is used to facilitate efficient batch processing. Additionally, the Bi-LSTM model is instantiated with defined parameters such as vocabulary size, embedding dimensions, and output dimensions.

3.3.2 Training Process

The training process follows these steps:

- **Device Configuration:** The model is configured to utilize a GPU if available, otherwise, it defaults to the CPU.

- **Model Definition:** The `TextClassificationModel` class is implemented with an embedding layer, an LSTM layer (bidirectional), and a fully connected layer.

- **Data Preparation:** Data is converted into tensor format, ensuring proper dtype compatibility.

Loss Function and Optimizer:

- The cross-entropy loss function is used, with class weights to handle imbalanced datasets.
- The Adam optimizer is selected for updating the model parameters efficiently.

Training Loop:

- The model is set to training mode.
- A loop runs for a predefined number of epochs.
- Each batch of data is fed into the model.
- The loss is calculated and backpropagated.
- The optimizer updates model weights

3.3.3 Evaluation Process

After training, the model is evaluated using a separate test dataset:

Inference Phase: The model is switched to evaluation mode (`model.eval()`).

The test dataset is loaded in batches.

Predictions are generated for each batch.

The loss and accuracy are computed.

Performance Metrics: Accuracy is computed by comparing predicted labels with true labels.

Validation loss and accuracy are printed.

3.4 Training and Validation Performance Analysis

4 Comparative Analysis with Other Neural Network Architectures

4.1 LSTM-Based Text Classification Model

The LSTM-based text classification model leverages Long Short-Term Memory (LSTM) networks to effectively capture sequential dependencies within text data. LSTM networks are widely used in natural language processing tasks due to their ability to retain information over long sequences, mitigating the vanishing gradient problem of traditional RNNs.

The model consists of:

- **Embedding Layer:** Converts input words into dense vector representations, improving feature learning.
- **LSTM Layer:** Processes sequences of word embeddings while preserving long-term dependencies.
- **Fully Connected Layer:** Uses the last hidden state from the LSTM to make classification predictions.
- **Weight Initialization:** Applies TensorFlow-like initialization using Xavier uniform for stable training.
- **Loss Function:** Utilizes `CrossEntropyLoss` with class weights to handle imbalanced datasets.
- **Optimizer:** Uses the Adam optimizer for efficient gradient updates.
- **Training and Validation Loops:** Iterates over multiple epochs, optimizing the model and evaluating performance.

Algorithm 2 Bi-LSTM Model Training and Evaluation

Require: Train dataset (X_{train}, Y_{train}) , Test dataset (X_{test}, Y_{test}) , Epochs E , Batch size B

Ensure: Trained Bi-LSTM model

```
Initialize Bi-LSTM model parameters
Set device: CUDA if available, else CPU
Convert dataset to tensor format
Create DataLoader for training and testing data
Define loss function: CrossEntropyLoss with class weights
Define optimizer: Adam with learning rate 0.001
for each epoch  $e$  in  $E$  do
    Set model to training mode
    Initialize loss and accuracy tracking
    for each batch  $(X_b, Y_b)$  in DataLoader do
        Move data to device
        Forward pass:  $outputs = model(X_b)$ 
        Compute loss:  $loss = criterion(outputs, Y_b)$ 
        Backpropagation: Compute gradients
        Update model weights using optimizer
        Track loss and accuracy
    end for
    Print training statistics (Loss, Accuracy)
end for
Set model to evaluation mode
Initialize validation loss and accuracy
for each batch  $(X_v, Y_v)$  in test DataLoader do
    Move data to device
    Forward pass:  $outputs = model(X_v)$ 
    Compute loss:  $loss = criterion(outputs, Y_v)$ 
    Track validation loss and accuracy
end for
Print validation statistics (Loss, Accuracy)
Return trained Bi-LSTM model
```

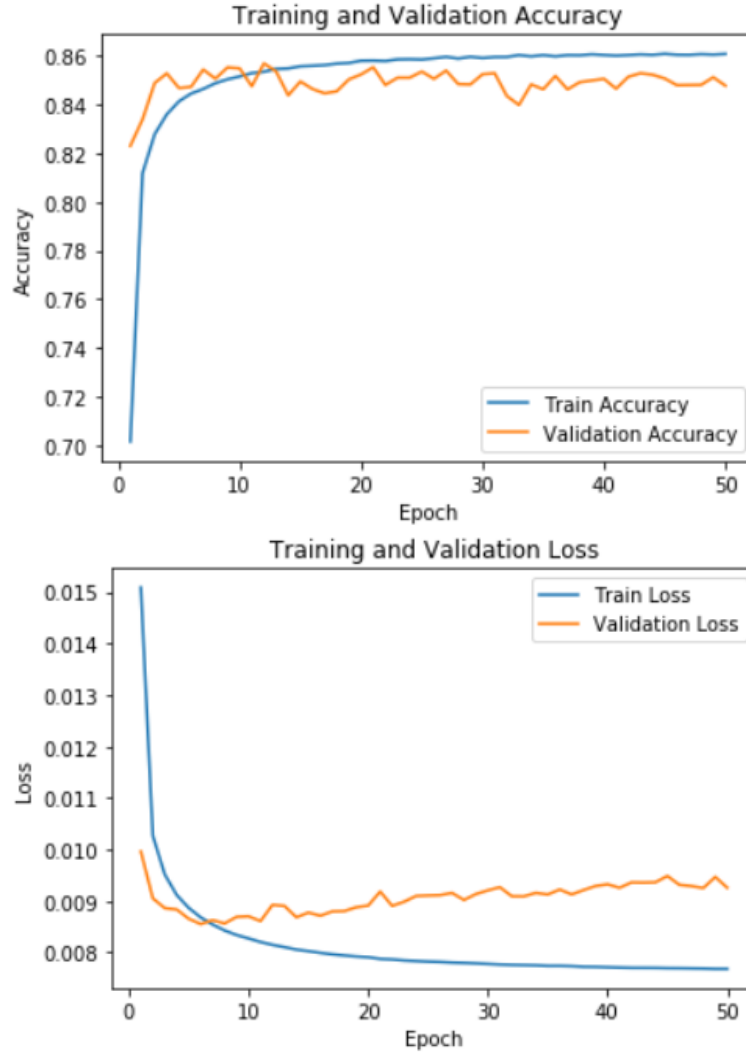


Figure 8: Training and Validation Performance Over 50 Epochs of Bi-LSTM Model. The top graph illustrates the accuracy trends over 50 epochs for both training and validation sets, showing a rapid initial improvement followed by convergence, with validation accuracy remaining consistently high. The bottom graph displays the corresponding loss curves, where the training loss decreases steadily while the validation loss stabilizes with minor fluctuations. Together, these plots indicate effective learning and generalization performance of the model.

Algorithm 3 LSTM Model Training and Evaluation

Require: Train dataset $(X_{\text{train}}, Y_{\text{train}})$, Test dataset $(X_{\text{test}}, Y_{\text{test}})$, Epochs E , Batch size B

Ensure: Trained LSTM model

```
Initialize LSTM model parameters
Set device: CUDA if available, else CPU
Convert dataset to tensor format
Create DataLoader for training and testing data
Define loss function: CrossEntropyLoss with class weights
Define optimizer: Adam with learning rate 0.001
for each epoch  $e$  in  $E$  do
    Set model to training mode
    Initialize loss and accuracy tracking
    for each batch  $(X_b, Y_b)$  in DataLoader do
        Move data to device
        Forward pass:  $outputs = model(X_b)$ 
        Compute loss:  $loss = criterion(outputs, Y_b)$ 
        Backpropagation: Compute gradients
        Update model weights using optimizer
        Track loss and accuracy
    end for
    Print training statistics (Loss, Accuracy)
end for
Set model to evaluation mode
Initialize validation loss and accuracy
for each batch  $(X_v, Y_v)$  in test DataLoader do
    Move data to device
    Forward pass:  $outputs = model(X_v)$ 
    Compute loss:  $loss = criterion(outputs, Y_v)$ 
    Track validation loss and accuracy
end for
Print validation statistics (Loss, Accuracy)
Return trained LSTM model
```

4.1.1 Performance Evaluation of LSTM Model



Figure 9: Training and Validation Performance Over 50 Epochs of LSTM Model. The top plot shows the training and validation accuracy over 50 epochs. The training accuracy increases rapidly and plateaus near 98%, while the validation accuracy stabilizes around 87%, indicating potential overfitting. The bottom plot illustrates the loss curves, where training loss decreases significantly and approaches zero, while validation loss increases gradually, further confirming that the model fits the training data very well but struggles to generalize effectively to the validation set.

4.2 RNN-Based Text Classification Model

The RNN-based text classification model leverages a simple recurrent neural network (RNN) architecture to process sequential text data and classify inputs into different categories. The model follows a structured workflow that includes defining the architecture, initializing model parameters, training the model using a loss function, and evaluating its performance on a validation dataset.

The model is designed with the following components:

- Embedding Layer: Converts words into dense vectors of fixed size.
- RNN Layer: Processes the sequence of word embeddings and captures temporal dependencies.

- Fully Connected Layer: Maps the output from the RNN to the number of output classes.

The training process involves converting text data into tensors, creating dataloaders for efficient batch processing, and optimizing the model using the Adam optimizer. During training, the model updates weights using backpropagation, and performance metrics such as loss and accuracy are tracked. The validation phase evaluates the model on unseen test data.

Algorithm 4 RNN Model Training and Evaluation

Require: Train dataset (X_{train}, Y_{train}) , Test dataset (X_{test}, Y_{test}) , Epochs E , Batch size B

Ensure: Trained RNN model

```

Initialize RNN model parameters
Set device: CUDA if available, else CPU
Convert dataset to tensor format
Create DataLoader for training and testing data
Define loss function: CrossEntropyLoss with class weights
Define optimizer: Adam with learning rate 0.001
for each epoch  $e$  in  $E$  do
    Set model to training mode
    Initialize loss and accuracy tracking
    for each batch  $(X_b, Y_b)$  in DataLoader do
        Move data to device
        Forward pass:  $outputs = model(X_b)$ 
        Compute loss:  $loss = criterion(outputs, Y_b)$ 
        Backpropagation: Compute gradients
        Update model weights using optimizer
        Track loss and accuracy
    end for
    Print training statistics (Loss, Accuracy)
end for
Set model to evaluation mode
Initialize validation loss and accuracy
for each batch  $(X_v, Y_v)$  in test DataLoader do
    Move data to device
    Forward pass:  $outputs = model(X_v)$ 
    Compute loss:  $loss = criterion(outputs, Y_v)$ 
    Track validation loss and accuracy
end for
Print validation statistics (Loss, Accuracy)
Return trained RNN model

```

4.2.1 Performance Evaluation of RNN Model

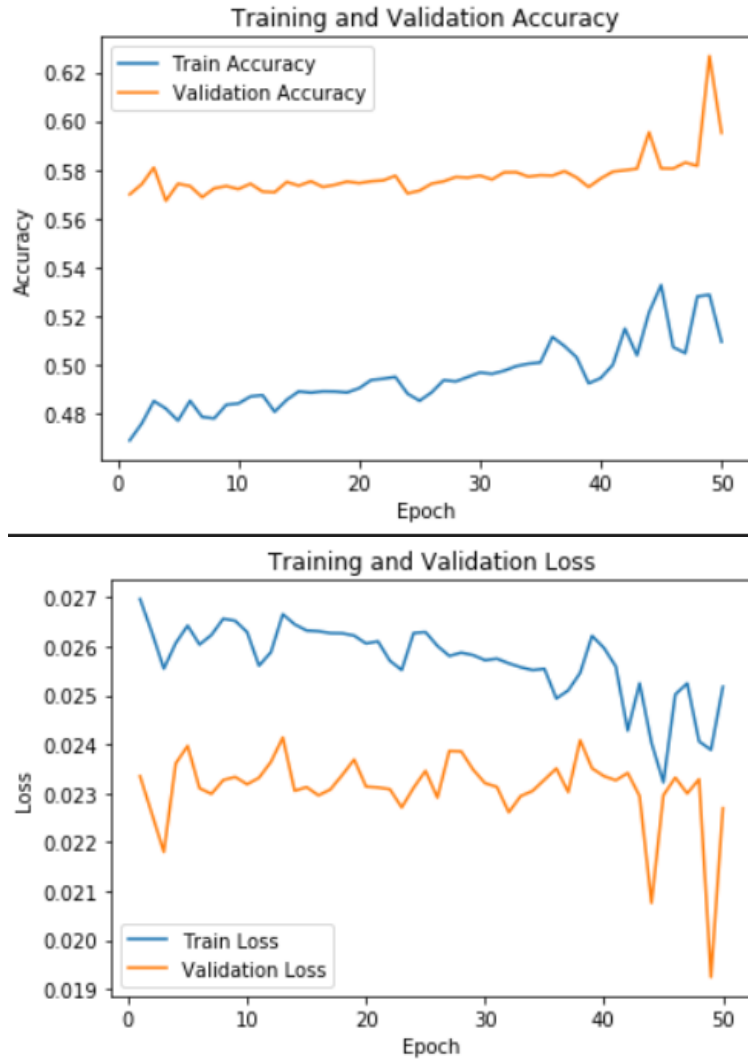


Figure 10: Training and Validation Performance Over 50 Epochs of RNN Model. The training and validation accuracy graph for the RNN model reveals that both accuracies remain significantly low throughout the 50 epochs, with training accuracy struggling to surpass 52% and validation accuracy hovering slightly above 60%. This indicates a poor learning capacity of the model. The loss graph further reinforces this, showing consistently high training and validation loss values with noticeable fluctuations, suggesting that the RNN model fails to effectively minimize error and capture the underlying data distribution.

4.2.2 Potential Problems Based on Poor Performances

- Vanishing Gradient Problem

RNNs suffer from vanishing gradients when dealing with long sequences, making it difficult to learn long-term dependencies. This results in the model failing to capture crucial patterns over time.

RNNs are weaker than LSTMs and GRUs in handling sequential data, particularly for complex text classification tasks. They struggle to remember dependencies over long text sequences.

- Slow and Ineffective Learning

The accuracy remains close to 50%, which is only slightly better than random guessing. The loss does not decrease significantly, suggesting that the model struggles to generalize.

- Better Alternatives Exist

Comparing with LSTM and Bi-LSTM models, this RNN architecture is outdated for text classification. Using LSTM or Transformer-based architectures could significantly enhance performance.

4.3 CNN-Based Text Classification Model

The CNN-based text classification model uses convolutional layers to extract spatial features from text sequences represented as embeddings. The model architecture includes:

- Embedding Layer: Converts words into dense vector representations.
- Convolutional Layers (Conv2D): Applies multiple convolutional filters of different sizes to extract n-gram features.
- Max-Pooling Layer: Reduces dimensionality by selecting the most significant features.
- Fully Connected (Dense) Layer: Maps the extracted features to output classes.
- Dropout Layer: Helps prevent overfitting by randomly deactivating neurons during training.
- The model is trained using CrossEntropyLoss with class weights to handle class imbalance and an Adam optimizer with a learning rate of 0.001.

Algorithm 5 CNN Model Training and Evaluation

Require: Train dataset (X_{train}, Y_{train}) , Test dataset (X_{test}, Y_{test}) , Epochs E , Batch size B

Ensure: Trained CNN model

Initialize CNN model parameters

Set device: CUDA if available, else CPU

Convert dataset to tensor format

Create DataLoader for training and testing data

Define loss function: CrossEntropyLoss with class weights

Define optimizer: Adam with learning rate 0.001

for each epoch e in E **do**

Set model to training mode

Initialize loss and accuracy tracking

for each batch (X_b, Y_b) in DataLoader **do**

Move data to device

Forward pass through convolutional layers and max-pooling

Compute loss: $loss = criterion(outputs, Y_b)$

Backpropagation: Compute gradients

Update model weights using optimizer

Track loss and accuracy

end for

Print training statistics (Loss, Accuracy)

end for

Set model to evaluation mode

Initialize validation loss and accuracy

for each batch (X_v, Y_v) in test DataLoader **do**

Move data to device

Forward pass through CNN model

Compute loss: $loss = criterion(outputs, Y_v)$

Track validation loss and accuracy

end for

Print validation statistics (Loss, Accuracy)

Return trained CNN model

4.3.1 Performance Evaluation of CNN Model



Figure 11: Training and Validation Performance Over 50 Epochs of CNN Model. The training and validation performance of the CNN model demonstrates a strong learning capability, as indicated by the training accuracy rapidly climbing to over 95% within the first 20 epochs. However, the validation accuracy stabilizes around 87%, suggesting potential overfitting. This trend is further supported by the loss graph, where training loss steadily decreases and approaches near-zero values, while validation loss gradually increases over time. The divergence between training and validation metrics highlights that while the CNN learns well on training data, its generalization to unseen data is limited.

4.4 Model Performance Comparison

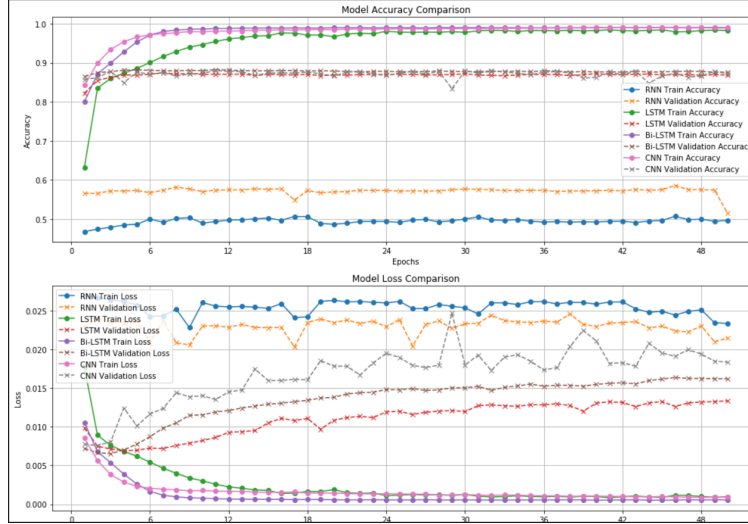


Figure 12: Performance Comparison. The performance comparison of RNN, LSTM, Bi-LSTM, and CNN models across 50 epochs shows significant differences in accuracy and loss. The top plot displays model accuracy, where Bi-LSTM consistently outperforms the other models, achieving nearly perfect accuracy on both training and validation sets. CNN and LSTM follow closely, while the standard RNN demonstrates the lowest accuracy due to its inability to handle long-term dependencies effectively. The bottom plot highlights the model loss, with Bi-LSTM achieving the lowest training and validation loss, indicating superior generalization. LSTM and CNN show moderate loss, while RNN exhibits high loss, reflecting poor performance and overfitting challenges. This comparison illustrates the advantage of Bi-LSTM in capturing sequential patterns and minimizing errors compared to other architectures.

5 Discussion on the future work of the neural network approach

The results of our experiments using neural networks for text classification indicate that while this approach shows great potential, it still faces a number of limitations that need to be addressed. Based on our evaluation of four distinct architectures—LSTM, Bi-LSTM, RNN, and CNN—we identify several directions for future work.

5.1 Model Performance Gap

The performance gap between training and validation metrics, particularly evident in the RNN and LSTM models, suggests issues with overfitting or insufficient generalization. While the LSTM model achieved a training accuracy exceeding 95%, its validation accuracy plateaued around 87%, indicating possible overfitting. [Sri+14] Regularization methods such as dropout, early stopping, or L2 regularization could be integrated more effectively to improve generalization. [GBC16] Additionally, fine-tuning hyperparameters or leveraging transfer learning techniques may help models generalize better to unseen data.

5.2 Data Imbalance

Despite using class-weighted loss functions were used to address class imbalance, performance skewed toward majority classes was still observed, particularly in the RNN and CNN models. This aligns with known challenges where neural networks tend to be biased toward dominant classes in imbalanced datasets [BMM18]. To mitigate this, future work could explore methods such as SMOTE (Synthetic Minority Oversampling Technique) [Cha+02], data augmentation, or cost-sensitive learning [ZL06]. Recent work also shows promise

in the application of generative adversarial networks (GAN) for the generation of synthetic data to balance minority classes[Fri+18].

5.3 Model Architecture

Architecture selection greatly influenced performance. Bi-LSTM outperformed the others, reflecting its strength in capturing both past and future dependencies in sequence data.[GS05] CNNs, although efficient in capturing local patterns, struggled with longer contextual relationships, as expected[Kim14]. RNNs performed worst, confirming known issues with vanishing gradients and limited memory capacity[BSF94]. To further improve performance, transformer-based models such as BERT[Dev+19] and attention mechanisms[Vas+17] could be explored, as they have recently demonstrated state-of-the-art performance in text classification.

6 Conclusion

In this thesis, a comparative study was conducted on the performance of four neural network architectures—LSTM, Bi-LSTM, CNN, and RNN—for multi-class text classification. Our findings revealed that while Bi-LSTM and LSTM models demonstrate promising results, especially in terms of accuracy and consistency, simpler architectures such as RNNs fail to provide competitive outcomes. The CNN model showed moderate performance, excelling in some cases but limited in generalizing sequential dependencies.

Despite these findings, the study highlights key challenges: overfitting, data imbalance, and architecture limitations. These challenges underline the complexity of deploying neural networks in real-world text classification tasks. Future research should focus on collecting more balanced datasets, incorporating additional features, and experimenting with more advanced architectures such as attention-based or transformer models.

This work provides a foundation for understanding how different neural models behave in a controlled setting. With further refinements, this line of research could substantially enhance the robustness and accuracy of automated text classification systems.

References

- [BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.
- [Cha+02] Nitesh V. Chawla et al. “SMOTE: Synthetic Minority Over-sampling Technique”. In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.
- [GS05] Alex Graves and Jürgen Schmidhuber. “Framewise phoneme classification with bidirectional LSTM and other neural network architectures”. In: *Neural Networks* 18.5-6 (2005), pp. 602–610.
- [ZL06] Zhi-Hua Zhou and Xu-Ying Liu. “Training cost-sensitive neural networks with methods addressing the class imbalance problem”. In: *IEEE Transactions on Knowledge and Data Engineering* 18.1 (2006), pp. 63–77.
- [Kim14] Yoon Kim. “Convolutional Neural Networks for Sentence Classification”. In: *EMNLP*. 2014. URL: <https://arxiv.org/abs/1408.5882>.
- [Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <https://www.deeplearningbook.org/>.
- [WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, et al. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *arXiv preprint arXiv:1609.08144* (2016).
- [Yan16] Shi Yan. *Understanding LSTM and its diagrams*. Accessed: [Insert Date]. 2016. URL: <https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714>.
- [Vas+17] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017. URL: <https://arxiv.org/abs/1706.03762>.
- [BMM18] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. “A systematic study of the class imbalance problem in convolutional neural networks”. In: *Neural Networks* 106 (2018), pp. 249–259.
- [Fri+18] Maayan Frid-Adar et al. “GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification”. In: *Neurocomputing* 321 (2018), pp. 321–331.
- [Dev+19] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *NAACL-HLT*. 2019. URL: <https://arxiv.org/abs/1810.04805>.
- [Ven19] P. Venugopal. “State-of-Health Estimation of Li-ion Batteries in Electric Vehicle Using IndRNN under Variable Load Condition”. In: *Energies* (Nov. 2019). DOI: [10.3390/en12224338](https://doi.org/10.3390/en12224338).
- [Vik22] Vikram. *Multiclass Complaints Classification Using Bi-LSTM*. Accessed: February 2024. 2022. URL: <https://www.kaggle.com/code/vikram92/multiclass-complaints-classification-using-bi-lstm>.
- [Iza+23] Alireza Izadi et al. “Control and diagnosis of brain tumors using deep neural networks”. In: *Proceedings of the 37th International Conference on Advanced Information Networking and Applications (AINA-2023), Volume 3*. May 2023.