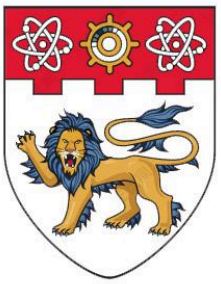


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC4001 CE/CZ4042: Neural networks and Deep Learning

Group Project

This project explores sentiment analysis on financial text using deep learning models, with a focus on handling small datasets and improving performance through vocabulary-aware preprocessing.

Gao Anni (N2402461C)

Zhang Yawen (N2402219G)

Apr 11 2025

Table of Contents

1. Introduction

One of the most significant applications of natural language processing (NLP) is sentiment analysis, which involves classifying the opinions or emotions expressed in text—typically into categories such as positive, negative, or neutral. In the financial domain, sentiment analysis plays a crucial role in supporting investment decisions and market analysis by extracting insights from news headlines, reports, and financial commentary.

In this project, we focus on the specific challenge of performing effective sentiment analysis on small, domain-specific datasets. Financial sentiment data, such as that in the publicly available FinancialPhraseBankdataset, is often limited in size and costly to annotate, which makes training high-performing deep learning models particularly difficult. This leads to problems such as overfitting, poor generalization, and inefficient use of model capacity, especially in embedding layers.

To explore how different architectures handle this limitation, we implement and evaluate four neural network models:

1. A standard LSTM model
2. An Attention-only model
3. A LSTM + Attention model
4. A Transformer + Attention model

The second part of this report introduces our improved model—a simple yet effective MLP architecture trained on the enhanced dataset. It outperforms the previous models in terms of validation accuracy and stability, demonstrating that with proper preprocessing, even smaller models can generalize well in low-resource NLP tasks.

2. Review of Existing Techniques

2.1 Dependencies

Install and import necessary libraries.

```
!pip install nltk
!pip install datasets
```

```python
import os
import datasets
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import nltk
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import accuracy_score

nltk.download('punkt')
nltk.download('punkt_tab')
torch.manual_seed(1)
```

2.2 Dataset Processing

Here, we explore the details of our dataset, prepare it for analysis, and extract key insights that will guide our modeling and evaluation in the subsequent stages of the project.

The dataset used in this part is the FinancialPhraseBank—a collection of carefully annotated financial news statements. Each sentence is labeled with a sentiment category (positive, neutral, or negative), making it suitable for training and evaluating deep learning models for sentiment analysis in the financial domain.

2.2.1 Load Data

We load the dataset using pandas.

```python
# Load the dataset. This is a sentiment analysis of financial texts.
data_pd = pd.read_csv('FinancialPhraseBank.csv', index_col=0)
data_pd.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4846 entries, 0 to 4845
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   text    4846 non-null   object
 1   label   4846 non-null   object
dtypes: object(2)
memory usage: 113.6+ KB
```

2.2.2 Text Preprocessing

To prepare the financial text data for sentiment analysis, we applied several preprocessing steps to ensure consistency and reduce noise:

- Lowercasing & Punctuation Removal: All text is converted to lowercase and cleaned of punctuation and special characters using regular expressions.
- Tokenization & Stopword Removal: Sentences are split into individual words using NLTK, and common English stopwords (e.g., “the”, “is”, “and”) are removed.
- Digit Filtering: Words containing numbers (like dates or percentages) are excluded to avoid irrelevant tokens.
- Label Encoding: Sentiment labels are mapped to integers—positive (2), negative (1), and neutral (0)—for model compatibility.

- Vocabulary Construction: After cleaning, a vocabulary is built from the remaining words. Each unique word is assigned an index, which allows the text to be converted into numerical inputs for the model.

```
import nltk
import re
from nltk import word_tokenize, sent_tokenize
from nltk.corpus import stopwords

nltk.download('stopwords')  # Download the stopwords data from NLTK
stops = stopwords.words("english")  # Store the list of English stopwords, which will be removed during preprocessing

## Check if a string contains any digits
def hasNumbers(inputString):
    return bool(re.search(r'\d', inputString))

## Regular expressions are a tool for describing and matching string patterns.
## They allow you to define a pattern and search for parts of the text that match it.
## For example: \d means any digit, \w means a letter, digit, or underscore.

## Function to preprocess text
def preprocess_txt_data(txt):
    ## Convert all characters to lowercase
    txt_lower = txt.lower()

    ## Remove all symbols and punctuation
    txt_lower = re.sub(r'[^\w\s]', '', txt_lower)  # Removes all non-alphanumeric and non-whitespace characters

    ## Tokenize the text into individual words
    all_words = word_tokenize(txt_lower)  # Split the cleaned text into tokens using word_tokenize

    cleaned_words = []
    for word in all_words:
        ## Remove stopwords and words containing digits
        if (word in stops) or (hasNumbers(word)):
            continue
        cleaned_words.append(word)

    return ' '.join(cleaned_words)

## Map sentiment labels to numeric values
label_map_dict = {'neutral': 0, 'negative': 1, 'positive': 2}
```

```
[nltk_data] Downloading package stopwords to /Users/ani/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
data_pd['cleaned_text'] = data_pd['text'].apply(lambda x: preprocess_txt_data(x))
data_pd['cleaned_label'] = data_pd['label'].apply(lambda x: label_map_dict[x])
```

```
## Build a task-specific vocabulary for this project, to be used for the word embedding module
all_words = []
for tmp_txt in data_pd['cleaned_text'].values:  # Use .values to access all text strings
    all_words += tmp_txt.split()  # Split each text string into a list of words and add them to all_words

alll = all_words  # Store all words (including duplicates)
all_unique_words = set(all_words)  # Convert the list to a set to remove duplicates and get all unique words

print("number of all:" , len(alll))
print('number of unique words: ', len(all_unique_words))

# Assign a unique index to each word
word_to_index = {}
idx = 0
for word in all_unique_words:
    word_to_index[word] = idx
    idx += 1
```

```
number of all: 57302
number of unique words:  9255
```

2.2.3 Dataset Preparation and Encoding

After preprocessing, we removed any samples with empty cleaned text. The dataset was then split into a training set (first 2500 samples) and a validation set (remaining samples). Each word in the cleaned text was mapped to its corresponding index based on the constructed vocabulary. We defined a custom PyTorch Dataset class to convert each sentence into a sequence of word indices and paired it with the corresponding sentiment label. Since the sentences have varying lengths, we used a batch size of 1 in the DataLoader to avoid shape mismatches. The model was set up to run on GPU if available, otherwise

on CPU.

```
data_pd = data_pd[data_pd.cleaned_text.apply(lambda x: len(x) > 0)]  # Remove samples with no valid characters

## In this experiment, we use the first 2500 samples as the training set and the remaining samples as the validation set
train_data_pd = data_pd.iloc[:2500]
val_data_pd = data_pd.iloc[2500:]
```

```
## Define a custom Dataset class for model training
class text_dataset(Dataset):
    def __init__(self, text_data, target, word_to_index_dict):
        super(text_dataset, self).__init__()
        self.inputs_x = []
        for tmp_txt in text_data:
            index_list = [word_to_index_dict[d] for d in tmp_txt.split()]
            self.inputs_x.append(np.array(index_list))
        self.inputs_y = target

    def __getitem__(self, index):
        tmp_x = self.inputs_x[index]
        tmp_y = self.inputs_y[index]
        return tmp_x, tmp_y

    def __len__(self):
        return len(self.inputs_x)
```

```
train_dataset = text_dataset(train_data_pd['cleaned_text'].values, train_data_pd['cleaned_label'].values, word_to_index)
val_dataset = text_dataset(val_data_pd['cleaned_text'].values, val_data_pd['cleaned_label'].values, word_to_index)

# Since sentence lengths vary, the input shapes differ. Therefore, batch training is not applicable
train_dataloader = DataLoader(train_dataset, batch_size=1, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=1, shuffle=False)
```

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

2.3 Model Training and Evaluation

2.3.1 LSTM-Based Classification Model

We implemented an LSTM-based classification model for sentiment analysis on financial text. The model consists of an embedding layer followed by an LSTM layer that captures sequential dependencies. The final hidden state is passed through linear layers for classification.

```
class LSTMClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMClassifier, self).__init__()
        # The model includes a word embedding layer that learns vector representations for each word during training
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        # The LSTM takes the word embeddings as input and outputs hidden states of dimension hidden_dim
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        # Apply a linear transformation to the LSTM output
        self.process_hidden = nn.Linear(hidden_dim, hidden_dim)
        # The final linear layer maps the hidden state space to the label space
        self.fc = nn.Linear(hidden_dim, tagset_size)  # classifier

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, (hideen_state, cell_state) = self.lstm(embeds)
        result = torch.tanh(self.process_hidden(lstm_out[:, -1, :]))  # Use the last hidden state for prediction
        result = self.fc(result)
        return result
```

Model 1: Architecture of the LSTM classifier. The final hidden state from the LSTM is passed through two linear layers to generate predictions.

2.3.2 Attention-Based Classification Model

We also implemented an attention-based classifier for sentiment analysis. The model maps each word to a dense vector using an embedding layer. An attention mechanism assigns weights to the word vectors, and a weighted average forms the sentence representation, which is then passed through a linear layer for classification.

```
## In the following model, after converting words into word embeddings,
## an attention module is applied to assign weights to each word vector,
## resulting in a single sentence-level representation. This final vector is used for prediction.
class AttentionClassifier(nn.Module):
    def __init__(self, embedding_dim, vocab_size, tagset_size):
        super(AttentionClassifier, self).__init__()
        # The model includes a word embedding layer, which learns vector representations of words during training
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        # The final linear layer maps the sentence vector to the label space
        self.fc = nn.Linear(embedding_dim, tagset_size)  # classifier
        self.w_d1 = nn.Linear(embedding_dim, embedding_dim)  # Transform word vectors first
        self.w_d2 = nn.Linear(embedding_dim, 1)  # Compute attention scores for each word

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        embeds_t = torch.tanh(self.w_d1(embeds))
        attention_score = torch.softmax(self.w_d2(embeds_t), dim=1)
        sentence_vector = torch.sum(attention_score * embeds_t, dim=1)
        result = self.fc(sentence_vector)  # Use the final sentence vector for prediction
        return result
```

Model 2: Architecture of the attention-based classifier. The model uses a weighted sum of transformed word embeddings to represent the input sentence for classification.

2.3.3 LSTM with Attention Model

We implemented a hybrid model that combines LSTM and attention mechanisms for sentiment classification. The LSTM layer captures the sequential structure and contextual relationships within the sentence, which is important for understanding financial text where word order matters. An attention mechanism is then applied to assign importance scores to each LSTM output, allowing the model to focus on the most relevant words. This combination leverages LSTM's ability to model context and attention's ability to highlight key information, making it more robust for complex sentences.

```
## Combine LSTM and Attention modules
class LSTMAttentionClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMAttentionClassifier, self).__init__()
        self.hidden_dim = hidden_dim
        # The model includes a word embedding layer, which learns vector representations of words during training
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        # The LSTM takes the word embeddings as input and outputs hidden states of dimension hidden_dim
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        # Apply a linear transformation to the output of the LSTM
        self.process_hidden = nn.Linear(hidden_dim, hidden_dim)
        # The final linear layer maps the hidden state space to the label space
        self.fc = nn.Linear(hidden_dim, tagset_size)  # classifier
        self.w_d1 = nn.Linear(hidden_dim, hidden_dim)  # Transform the hidden states before attention
        self.w_d2 = nn.Linear(hidden_dim, 1)  # Compute attention weights for each hidden state

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, (hideen_state, cell_state) = self.lstm(embeds)
        ## Apply attention over all hidden states from the LSTM to generate a sentence representation
        lstm_out_t = torch.tanh(self.w_d1(lstm_out))
        attention_score = torch.softmax(self.w_d2(lstm_out_t), dim=1)
        sentence_vector = torch.sum(attention_score * lstm_out_t, dim=1)
        result = self.fc(sentence_vector)  # Use the final sentence vector for prediction
        return result
```

Model 3: Attention mechanism applied on top of all LSTM outputs. Each hidden state is scored and weighted to compute the final sentence representation used for classification.

2.3.4 Transformer with Attention Model

We implemented a Transformer-based sentiment classifier with an added attention mechanism. The Transformer encoder captures global context across the entire sentence using multi-head self-attention, while the additional attention layer helps the model focus on the most informative tokens for classification. By combining these two components, the model benefits from both deep contextual understanding and targeted emphasis on keywords, making it suitable for complex financial text.

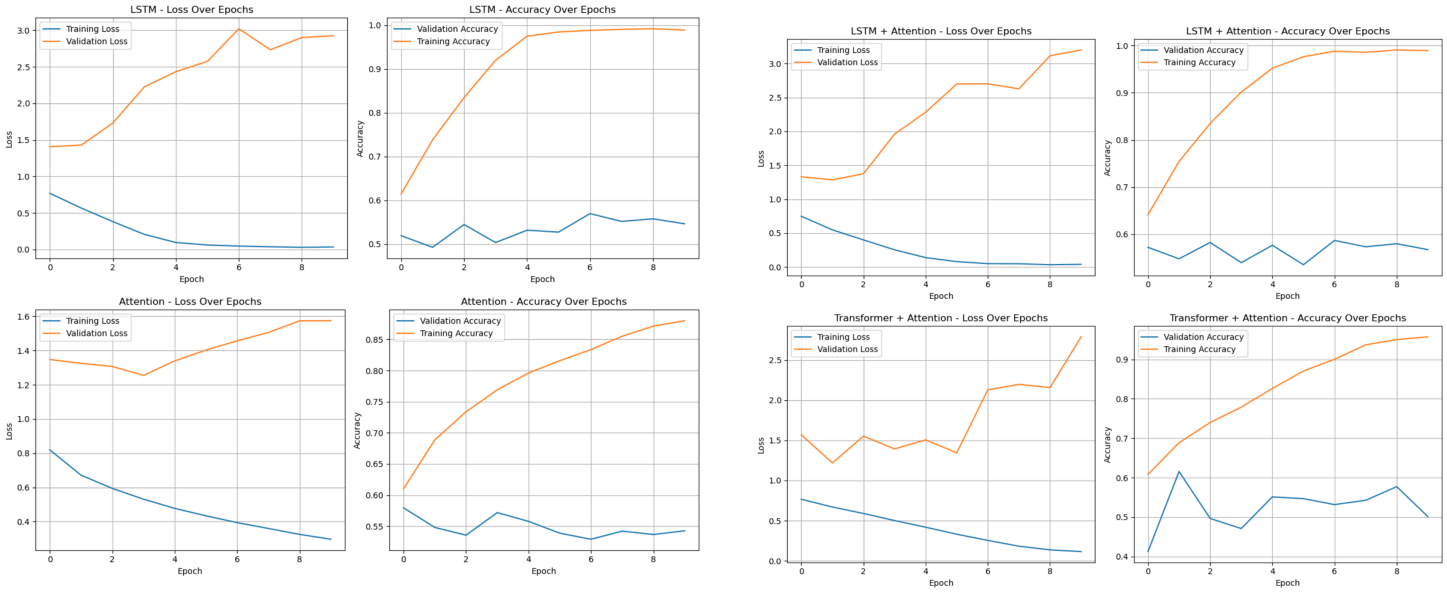
```python
## Combine the Transformer Encoder module and an additional Attention module
class TransformerAttentionClassifier(nn.Module):
    def __init__(self, embedding_dim, vocab_size, tagset_size, max_seq_length=1000):
        super(TransformerAttentionClassifier, self).__init__()
        # The model includes a word embedding layer, which learns vector representations of words during training
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        # Add position embeddings (trainable)
        self.position_embeddings = nn.Embedding(max_seq_length, embedding_dim)
        self.encoder = nn.TransformerEncoderLayer(d_model=embedding_dim, nhead=8)
        # Apply a linear transformation to the encoder output
        self.process_hidden = nn.Linear(embedding_dim, embedding_dim)
        # Final linear layer maps hidden states to the label space
        self.fc = nn.Linear(embedding_dim, tagset_size)  # classifier
        self.w_d1 = nn.Linear(embedding_dim, embedding_dim)  # Transform word vectors before computing attention
        self.w_d2 = nn.Linear(embedding_dim, 1)  # Compute attention weights for each token

    def forward(self, sentence):
        word_embeds = self.word_embeddings(sentence)
        # Generate position IDs (automatically adapted to input length)
        positions = torch.arange(0, sentence.size(1), dtype=torch.long, device=sentence.device)
        positions = positions.unsqueeze(0).expand_as(sentence)  # (batch_size, seq_len)
        # Add position embeddings
        pos_embeds = self.position_embeddings(positions)  # (batch_size, seq_len, embedding_dim)
        embeds = word_embeds + pos_embeds

        encoder_output = self.encoder(embeds)
        ## Apply attention to all encoder outputs to compute the final sentence vector
        encoder_output_t = torch.tanh(self.w_d1(encoder_output))
        attention_score = torch.softmax(self.w_d2(encoder_output_t), dim=1)
        sentence_vector = torch.sum(attention_score * encoder_output_t, dim=1)
        result = self.fc(sentence_vector)  # Use the sentence vector for final prediction
        return result
```

Model 4: Input tokens are embedded and combined with learnable positional embeddings, then passed into a Transformer encoder layer to capture global context. A second attention layer computes token-level weights to generate the final sentence representation.

2.3.5 Model Comparison

We trained and evaluated four deep learning architectures for sentiment analysis on financial text: LSTM, Attention-only, LSTM with Attention, and Transformer with Attention. Each model's performance was monitored across 10 epochs using training and validation loss and accuracy.

Based on the training and validation curves of all four models—LSTM, Attention, LSTM + Attention, and Transformer + Attention—we observe a common trend: overfitting.



As shown in the plots, all models achieve steadily decreasing training loss and increasing training accuracy. However, validation loss either remains flat or worsens over time, and validation accuracy fluctuates or stays consistently lower than expected. This indicates that the models are memorizing the training data but failing to generalize to unseen examples.

- The LSTM-based models are particularly prone to overfitting. They quickly reach near-perfect accuracy on the training set but exhibit unstable and low validation accuracy, suggesting poor generalization.
- Attention-based models demonstrate slightly more stable performance. Notably, the Transformer + Attention model, despite having lower training accuracy, maintains more consistent validation results. This suggests that attention mechanisms are more robust under low-data settings by helping the model focus on key tokens instead of memorizing sequences.

These results reveal a key limitation: the dataset is too small and sparse for deep models to generalize well, especially when trained from scratch without leveraging external knowledge.

3. Problem Identification - Vocabulary Sparsity in Small Data

During preprocessing, we observed that the dataset contains 57302 words, 9255 unique words. After splitting the dataset into training and validating, we count the number and unique number in each dataframe. As the numbers show in the table, we find that some words are not overlapping in the 2 dataframe, which means some words might never appear in the training set at all but do appear in the validation set. This leads to vocabulary sparsity in which many words appear very infrequently, possibly

just once or twice in the training set, or only in the validation set. As a result, the nn.Embedding layer may learn inefficient or unreliable word embeddings for these rare or unseen words.

4. Proposed Solution

To address this, we manipulate the dataset by the following strategies:

1. Count each word's appearing frequency in both datasets and find the low frequent words or words that only appear in the validation set. The number of words is counted to be 2717.
2. Define a function to use “unknown” to replace these low frequent words and apply the new version as the new column to be trained and tested.
3. Used text preprocessing technique PorterStemmer() function to reduce words to their base or root form, increasing the generalization, reduces vocabulary size and improves learning efficiency by combining semantically similar words.
4. Update the 2 datasets, create new lists and a new dictionary to store the manipulated words.
5. Recount the number of low frequent words to check the usefulness of the techniques, and the output number is only 1 word, indicating a well-structured framework.

4.1 Improved Model Implementation and Results

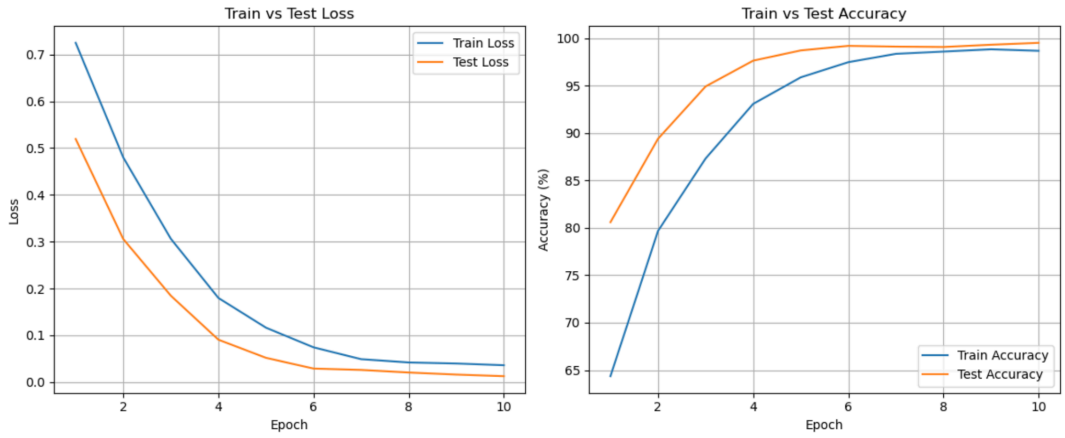
The new version model uses an MLP with the manipulated data shown in the figure, with embed_dim = 200, hidden_dim = 200, and drop_out=0.2, and the result is obviously better than the previous models. The model uses an nn.Sequential container to define a feedforward network:

- A linear transformation from the embedding dimension to a hidden dimension (nn.Linear(embed_dim, hidden_dim)),
- A ReLU activation to introduce non-linearity,
- A dropout layer to prevent overfitting by randomly dropping some neurons during training,
- A final linear layer projecting to the output dimension, which corresponds to the number of classes in the classification task.

In the forward pass, the model first applies the embedding layer to the input tensor x, which typically has the shape (batch_size, sequence_length). This operation transforms each word in the sequence into its corresponding embedding vector, resulting in a tensor of shape (batch_size, sequence_length, embed_dim). To obtain a fixed-size sentence representation, the model performs average pooling across the sequence length dimension. Specifically, it computes the mean of all word embeddings in the sentence using x.mean(dim=1), resulting in a tensor of shape (batch_size, embed_dim). This pooled representation is then passed through the feedforward network (self.net), producing an output of shape (batch_size, output_dim). This output can then be used for classification tasks such as sentiment analysis or topic prediction.

```
: class MLP(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim, drop_out=0.2):
        super(MLP, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.net = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(drop_out),
            nn.Linear(hidden_dim, output_dim)
            # nn.Linear(embed_dim, output_dim)
        )

    def forward(self, x):
        x = self.embedding(x)
        x = x.mean(dim=1)
        out = self.net(x)
        return out
```

```
Epoch [1/10] | Train Loss: 0.7206, Train Acc: 64.28% | Test Loss: 0.4933, Test Acc: 82.04%
Epoch [2/10] | Train Loss: 0.4574, Train Acc: 80.48% | Test Loss: 0.2704, Test Acc: 91.04%
Epoch [3/10] | Train Loss: 0.2668, Train Acc: 89.16% | Test Loss: 0.1269, Test Acc: 96.80%
Epoch [4/10] | Train Loss: 0.1318, Train Acc: 95.44% | Test Loss: 0.0678, Test Acc: 97.92%
Epoch [5/10] | Train Loss: 0.0781, Train Acc: 97.72% | Test Loss: 0.0690, Test Acc: 98.36%
Epoch [6/10] | Train Loss: 0.0510, Train Acc: 98.36% | Test Loss: 0.0185, Test Acc: 99.52%
Epoch [7/10] | Train Loss: 0.0345, Train Acc: 99.20% | Test Loss: 0.0791, Test Acc: 98.08%
Epoch [8/10] | Train Loss: 0.0416, Train Acc: 98.76% | Test Loss: 0.0209, Test Acc: 99.40%
Epoch [9/10] | Train Loss: 0.0270, Train Acc: 99.12% | Test Loss: 0.0142, Test Acc: 99.48%
Epoch [10/10] | Train Loss: 0.0297, Train Acc: 99.08% | Test Loss: 0.0189, Test Acc: 99.52%
```



- There's no sign of overfitting, and test accuracy is very close to or even higher than training accuracy.
- Test loss continues to drop, with no plateau or spike, which means the model is still improving.

5. Conclusion

In this project, we explored sentiment analysis on financial text using deep learning models under a small-data constraint. After implementing and evaluating four baseline models—LSTM, Attention, LSTM + Attention, and Transformer + Attention—we found that all of them showed signs of overfitting. This was

especially evident in the gap between training and validation accuracy, which pointed to poor generalization.

Further analysis revealed that vocabulary sparsity was a major issue. The dataset included many rare or unseen words, which made it difficult for the embedding layers to learn reliable representations. To address this, we introduced a data preprocessing strategy that replaced low-frequency words with an <unknown> token and used stemming to consolidate similar word forms.

Using this enhanced dataset, we trained a simplified MLP-based model. Despite its smaller architecture, the model achieved much better validation accuracy and stability, showing no overfitting and continual improvement throughout training. These results demonstrate that in low-resource NLP tasks, a well-designed preprocessing pipeline can be more effective than simply increasing model complexity.

Dataset Citation

```
@article{Malo2014GoodDO,
  title={Good debt or bad debt: Detecting semantic orientations in economic texts},
  author={P. Malo and A. Sinha and P. Korhonen and J. Wallenius and P. Takala},
  journal={Journal of the Association for Information Science and Technology},
  year={2014},
  volume={65}
}
```