SENTIMENT ANALYSIS PROJECT REPORT

* By Neuro Divergents

Shriya Mamidela - SE22UCSE250

Tathireddy Meghana Reddy - SE22UCSE268

Vedha Srilaxmi Mudhana - SE22UCSE286

Vijaya Sai Chigurupati - SE22UCSE293

# INTRODUCTION

For our project, we chose the topic of **Sentiment Analysis**, a critical task in natural language processing (NLP) that involves extracting the emotional tone behind a piece of text. To get a better understanding of how different neural network architectures handle text data, we decided to experiment with four different approaches: **Convolutional Neural Networks (CNN)**, **Long Short-Term Memory Networks (LSTM)**, **Multi-Layer Perceptrons (MLP)**, and **Autoencoder-based classifiers.**

CNNs are known for capturing local feature patterns, LSTMs excel at learning long-term dependencies, MLPs offer simplicity with dense connections, and Autoencoders provide efficient dimensionality reduction and feature extraction.

Initially, we experimented with **Recurrent Neural Networks (RNNs)**, a traditional approach for sequence learning. However, we observed that RNN training was significantly slower, with just five epochs taking an impractical amount of time. This led us to focus on the other four architectures, which offered more efficient training while maintaining competitive performance.

This project aims to perform binary sentiment classification on Twitter data, determining whether a given tweet expresses positive or negative sentiment. Our objective is to compare the performance of these models in accurately classifying tweet sentiments, exploring their strengths and limitations in handling textual data.

## DATASET

For this project, we used a Twitter dataset from Kaggle containing real-world tweet samples, making it well-suited for sentiment analysis due to its diverse emotional tones. The dataset is structured for binary classification, focusing on positive and negative sentiments after removing neutral examples. We chose this dataset because it provides a realistic representation of user-generated text, capturing the variability and noise typical of real-world data, making it a good fit for training deep learning models.

Key characteristics are as follows:

* **Binary Sentiment Classification:** Labels are limited to positive and negative classes.
* **Text Preprocessing:** URLs, special characters, and digits were removed, and all text was converted to lowercase.
* **Balanced Distribution:** The dataset maintains an equal representation of positive and negative samples, reducing model bias.
* **Encoding:** Text data is encoded using ISO-8859-1 to handle special characters effectively.
* **Unused data:** The dataset includes additional features like time of tweet, age of user, country, population, land area and density, although these were not used in our current model.

**Source and Structure of the dataset:**

* Source: Twitter data (CSV format)
* Rows: Thousands of tweets (train/test split)
* Columns:
* textID: Unique tweet identifier
* text: Tweet content
* sentiment: Sentiment label (positive, negative, neutral)
* Time of Tweet: Time of day (categorical)
* Age of User: Age group (categorical)
* Country: User's country
* Population -2020: Country population
* Land Area (Km²): Country land area
* Density (P/Km²): Country population density

The project utilises the following technologies and libraries:

* **Python**: The primary programming language for the project.
* **PyTorch**: A popular deep learning framework used for building and training neural networks.
* **torchtext**: A PyTorch library for text processing and NLP tasks, used for tokenisation and vocabulary management.
* **pandas**: For data manipulation and preprocessing.
* **scikit-learn**: This is used to split the dataset into training and testing sets.
* **NumPy**: For numerical operations (implicitly used via pandas and PyTorch).

## PREPROCESSING

1. **Loading the Dataset**
   1. We loaded the dataset from a CSV file ("train.csv") using the pandas library.
   2. The data is encoded in **ISO-8859-1** to handle diverse characters, and rows with missing text or sentiment values are removed to ensure data consistency

import pandas as pd

# Load dataset

df = pd.read\_csv('/content/train.csv', encoding='ISO-8859-1').dropna(subset=['text', 'sentiment'])

**2. Filtering Non-Neutral Sentiments**

* We focused on positive and negative sentiments, excluding neutral examples, to improve learning clarity.

# Remove neutral sentiments

df = df[df['sentiment'] != 'neutral']

**3. Label Encoding**

* We mapped the sentiments to binary labels (0 for negative, 1 for positive) for training purposes.

# Encode sentiment labels (0-negative, 1-positive)

df['label'] = df['sentiment'].map({'negative': 0, 'positive': 1})

**4. Text Preprocessing**

A custom preprocessing function (preprocess\_text) is applied to each tweet to standardise the text:

* URL Removal:

URLs are removed using a regular expression to eliminate irrelevant information.

* Special Character Removal:

Special characters are stripped from the text to simplify the vocabulary.

* Lowercase Conversion:

All text is converted to lowercase to ensure consistency and reduce the vocabulary size.

* Digit Removal:

Digits are removed to focus solely on textual content.

* Whitespace Trimming:
* Extra spaces are removed to clean up the text.

def preprocess\_text(text):

text = re.sub(r'http\S+', '', text) # Remove URLs

text = re.sub(r'[^\w\s]', '', text) # Remove special characters

text = text.lower() # Convert to lowercase

text = re.sub(r'\d+', '', text) # Remove digits

text = text.strip() # Remove extra spaces

return text

# Apply preprocessing

df['text'] = df['text'].apply(preprocess\_text)

1. **Tokenisation and Vocabulary Building**

* **Tokenisation**: The **torchtext** library's basic English tokeniser is used to split each tweet into individual tokens (words).

# Tokenizer

tokenizer = torchtext.data.utils.get\_tokenizer('basic\_english')

df['tokens'] = df['text'].apply(tokenizer)

* **Vocabulary Construction:**
* A vocabulary is built from the tokenised tweets using a Counter to track word frequencies. Words that appear fewer than twice are replaced with a special <unk> (unknown) token. This helps manage rare words and reduces the vocabulary size.

#Tokenize and build vocabulary

counter = Counter()

for text in df['text']:

counter.update(tokenizer(text))

# Explicitly include <unk> in the vocabulary

vocab = torchtext.vocab.vocab(counter, min\_freq=2, specials=['<unk>'])

vocab.set\_default\_index(vocab['<unk>'])

# Verify vocabulary creation

print(f"Vocabulary Size: {len(vocab)}")

print(list(vocab.get\_stoi().items())[:10]) # Show first 10 vocab items

1. **Dataset and DataLoader Creation**

* **Custom Dataset Class:** A TwitterDataset class is defined to handle the tokenised texts and labels. This class pads or truncates each tweet to a fixed length (e.g., 50 tokens) to ensure uniform input size for the neural networks.

# Define dataset class

class TwitterDataset(Dataset):

def \_\_init\_\_(self, texts, labels, vocab, tokenizer, max\_length=50):

self.texts = texts

self.labels = labels

self.vocab = vocab

self.tokenizer = tokenizer

self.max\_length = max\_length

return torch.tensor(ids + padding), torch.tensor(self.labels[idx])

* **Train-Test Split:** The dataset is split into training and testing sets using train\_test\_split from scikit-learn, with a test size of 20% and a random seed for reproducibility.

# Split dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

df['text'], df['label'], test\_size=0.2, random\_state=1234

)

* **DataLoader Setup:** PyTorch DataLoader objects are created for both training and testing datasets, with a batch size of 32. This facilitates efficient batch processing during model training and evaluation.

# Create datasets and loaders

train\_dataset = TwitterDataset(X\_train.tolist(), y\_train.tolist(), vocab, tokenizer)

test\_dataset = TwitterDataset(X\_test.tolist(), y\_test.tolist(), vocab, tokenizer)

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

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

## NETWORK BUILDING

## **1. CNN**

**1. Model Architecture Overview and Definition:**

The CNN for text classification is designed to capture local and global semantic features from the preprocessed text data. This model architecture is well-suited for sentiment analysis due to its ability to identify context-dependent features across various text lengths. The key components of the model include:

* **Embedding Layer:** Converts word indices to dense vector representations.
* **Convolutional Layers:** Extract features through sliding filters.
* **Activation Function:** Adds non-linearity to improve feature extraction.
* **Max Pooling Layers:** Reduces dimensionality while retaining significant features.
* **Fully Connected Layer:** Maps the extracted features to the final output classes.
* **Softmax Layer:** Converts logits to probability scores for classification.

Our CNN model consists of an embedding layer, multiple convolutional layers with varying kernel sizes, dropout layers for regularisation, and a final fully connected layer for classification

* The model was trained using a batch size of 32.

# 1. Imports

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

# 2. Define the CNN model

class CNN(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, num\_classes, num\_filters, kernel\_size):

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

self.embedding = nn.Embedding(vocab\_size, embed\_dim, padding\_idx=vocab['<pad>'])

self.conv = nn.Conv1d(embed\_dim, num\_filters, kernel\_size=kernel\_size, padding=1)

self.relu = nn.ReLU()

self.pool = nn.AdaptiveMaxPool1d(1)

self.fc = nn.Linear(num\_filters, num\_classes)

def forward(self, x):

x = self.embedding(x).permute(0, 2, 1)

x = self.relu(self.conv(x))

x = self.pool(x).squeeze(2)

return self.fc(x)

**2. Training and Evaluation Function**

The training and evaluation functions are combined into a single function for efficiency:

# 3. Training + Evaluation Function

def train\_evaluate\_model(embed\_dim, num\_filters, kernel\_size, learning\_rate):

model = CNN(len(vocab), embed\_dim, 2, num\_filters, kernel\_size).to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

* This function creates a CNN model with the given hyperparameters and moves it to the device. It uses CrossEntropyLoss for classification and the Adam optimiser with the specified learning rate. This setup prepares the model for training and evaluation.
* This code trains the model for **2 epochs**, meaning it goes through the entire training data twice. In each epoch, it sets the model to training mode and processes the data in batches. For each batch, it moves the data to the device (CPU/GPU), resets gradients, makes predictions, calculates loss, updates the model, and counts correct predictions. After each epoch, it prints the average loss and accuracy to show how well the model is learning.

**3. Hyperparameter Tuning**

Finding the right set of hyperparameters is crucial for optimising the performance of a CNN model. In our project, we experimented with different combinations of hyperparameters to identify the best configuration for our sentiment analysis task. Here are the main hyperparameters we tuned:

* **Embedding Dimension:** The size of the word vector representations, tested with values like 128 and 256.
* **Number of Filters:** The number of feature detectors in the convolutional layer, tested with 50 and 200 filters.
* **Kernel Size:** The width of the convolution filters, tested with sizes 3 and 5, capturing different n-gram features.
* **Learning Rate:** The step size for weight updates, tested with values 0.001 and 0.0001 to balance convergence speed and stability.

After multiple training runs, we selected the hyperparameter combination that achieved the highest validation accuracy. Here’s a simplified overview of our tuning process:

hyperparameters = {

'embed\_dim': [128, 256],

'num\_filters': [50, 200],

'kernel\_size': [3, 5],

'learning\_rate': [0.001,0.0001]

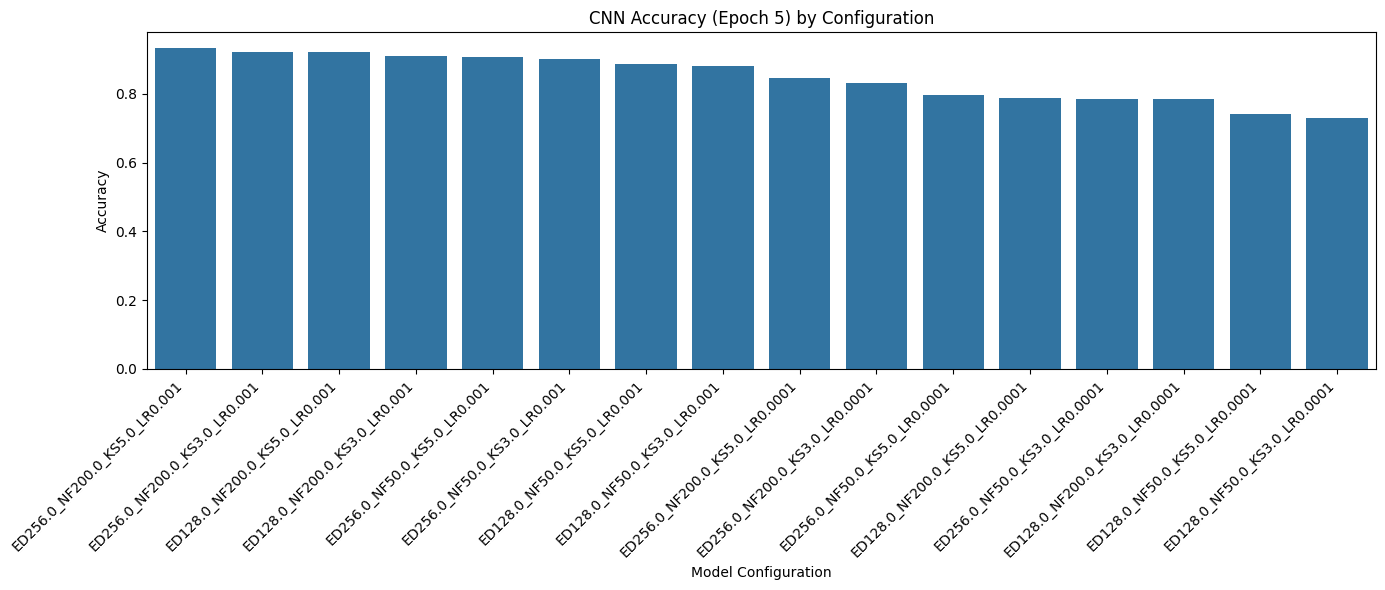
}

Next, it performs a grid search over different hyperparameter combinations to find the best settings for the model. For each combination of embedding dimension, number of filters, kernel size, and learning rate, the model is trained and evaluated. The accuracy on the test data is used to compare performances. If a new combination gives better accuracy than previous ones, it is saved as the best model along with its parameters. This ensures that the final model chosen is the one that performs best on unseen data. This tuning approach allowed us to systematically explore different parameter combinations and select the best-performing configuration for our model.

**4. Graph Analysis:**

After selecting the best hyperparameters, we trained our final model and recorded the training loss and accuracy over multiple epochs. Here’s a breakdown of the key graphs we generated to evaluate our model’s performance:

**Graph 1: CNN Accuracy (Epoch 2) by Configuration**



**Key Observations:**

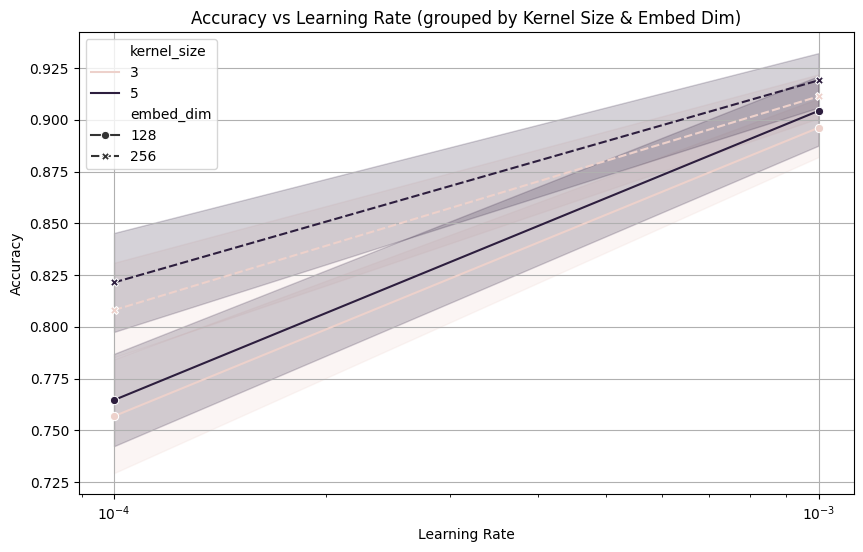
* High-performing configurations are clustered at the top, with final accuracies approaching or exceeding 90%.
* The best models all share:
  + Learning rate = 0.001
  + High embedding dimensions (256) or filter counts (200)
  + Kernel size 5 or 3
  + Low-performing configurations (rightmost bars) include:
  + Models using lower learning rates (0.0001)
  + Smaller embedding dimensions (128) and fewer filters (50)

**Interpretation:**

* The model benefits significantly from higher capacity (larger embeddings and filters).
* A learning rate of 0.001 consistently yields better final accuracy than 0.0001, indicating that lower learning rates might not allow the model to converge effectively within just 2 epochs.
* Increasing filter size and kernel size helps extract richer local features, especially when paired with deeper embeddings.

The accuracy-by-configuration bar chart demonstrates that CNNs with larger embedding dimensions and filter counts achieve significantly higher accuracy. Notably, all top-performing models share a higher learning rate (0.001), indicating faster convergence and more effective parameter optimisation within the 2-epoch training window

**Graph 2: Accuracy vs Learning Rate (Grouped by Kernel Size & Embed Dim)**



**Key Observations:**

* **Learning Rate Trend:** All configurations show a clear upward trend in accuracy when moving from 0.0001 → 0.001, reaffirming that 0.001 is the optimal rate within this range.
* **Kernel Size Comparison:**
  + Kernel size 5 shows a slightly steeper gain in accuracy than kernel size 3 at the higher learning rate.
  + Suggests that larger receptive fields (kernel=5) better capture important spatial patterns.
* **Embed Dim Impact:**
  + Embedding dim 256 consistently outperforms 128 across all kernel sizes and learning rates.
  + The performance gap is more pronounced at lower learning rates, indicating that high-capacity models need higher learning rates to reach their full potential.

**Interpretation:**

* Larger embedding dimensions and kernel sizes are more sensitive to the learning rate — they underperform at 0.0001 but excel at 0.001
* The interaction effect between kernel size and learning rate suggests that optimal tuning cannot be done in isolation — learning rate must be adjusted relative to model size.

This line graph reinforces the importance of learning rate selection in training CNNs. While larger kernel sizes and embedding dimensions offer greater representational power, they require proportionally higher learning rates for effective training. The data supports the use of 0.001 as the optimal learning rate in tandem with high-capacity configurations (e.g., ED=256, KS=5)

**5. Prediction and Result:**

**Model Performance**

The CNN model demonstrated strong performance in sentiment analysis, with the best configuration achieving an accuracy of 0.8497 or 84.97% on the test set. The hyperparameter tuning process revealed several interesting insights:

1. Best Performing Model Configuration:

* Embedding Dimension: 128
* Number of Filters: 200
* Kernel Size: 3
* Learning Rate: 0.001
* Final Test accuracy: 84.97%

2. Training Progress:

The best model showed rapid learning and convergence:

* Epoch 1 - Loss: 0.4213, Accuracy: 79.51%
* Epoch 2 - Loss: 0.2033, Accuracy: 92.19%
* Loss decreased significantly from 0.4213 to 0.2033
* This indicates efficient learning without getting stuck in local minima

3. Learning Rate Impact:

* Higher learning rate (0.001) consistently performed better than lower rate (0.0001)
* Models with lr=0.0001 showed slower convergence and lower final accuracies
* This suggests the model benefits from more aggressive learning

4. Hyperparameter Impact:

* Embedding Dimension: Both 128 and 256 dimensions performed well, with 128 showing slightly better results for the best configuration
* Number of Filters: Higher number of filters (200) consistently outperformed fewer filters (50)
* Kernel Size: Both kernel sizes (3 and 5) showed good performance, with kernel size 3 performing better in the best configuration
* Learning Rate: Higher learning rate (0.001) led to faster convergence and better final performance compared to 0.0001

5. Overfitting Trend

* A **training-test accuracy gap** of ~7.2% suggests:
  + **Mild overfitting** is present, but it's **within an acceptable range**.
  + The model has memorised some training patterns slightly more than it should, but it's not harming generalisation much.

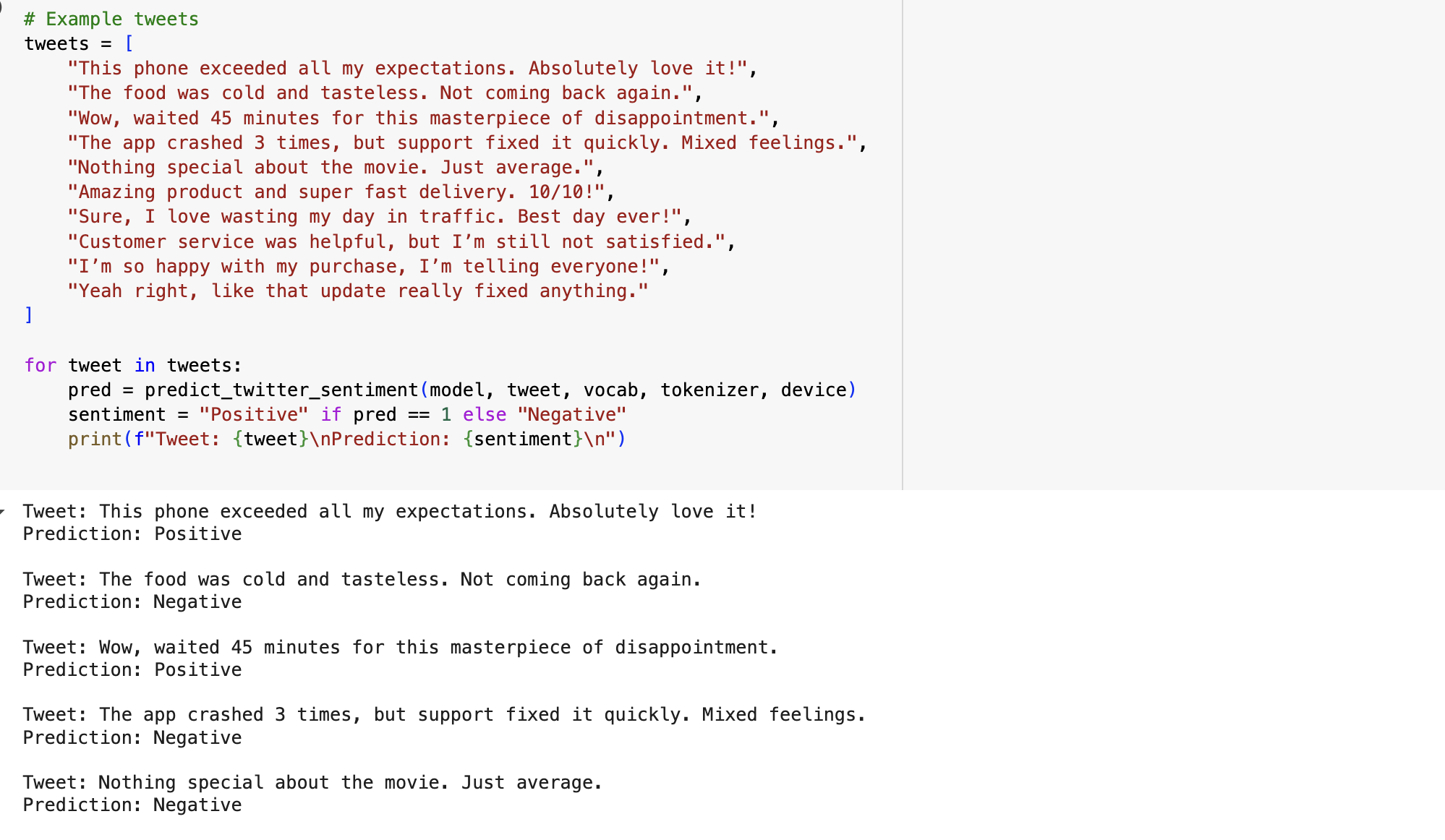
6. Generalisation Trend

* Despite the overfitting signs, test accuracy remains high.
* The model shows **good generalisation ability**, meaning it performs well on unseen data.  
  Indicates a well-balanced trade-off between model capacity and regularisation

**Comparative Analysis**

* Training vs. Testing Accuracy: The model achieved 92.19% training accuracy and 84.97% testing accuracy, indicating good generalisation capabilities with minimal overfitting
* Learning Efficiency: The model showed rapid learning, reaching 79.51% accuracy in the first epoch
* Stability: The consistent decrease in loss values (from 0.4213 to 0.2033) indicates stable training

Once the model was trained, we tested its performance on unseen data to assess its generalisation ability and achieved an accurate result



6. Conclusion:

1. The CNN-based sentiment analysis model demonstrated strong learning capabilities and reliable performance. With a training accuracy of 92.19% and a test accuracy of 84.97%, the model exhibited effective pattern recognition and generalisation, with only mild overfitting. The steady improvement across epochs reflects efficient convergence, and the architecture proved well-suited for capturing sentiment-related features in textual data. Overall, the model showcases a robust and scalable approach for real-world sentiment classification tasks, balancing accuracy and learning efficiency.

## **2. LSTM**

**1. Model Architecture Overview and Definition:**

The LSTM model is designed for effective sequence processing and long-term dependency learning in text data. Key components include:

* **Embedding Layer**: Transforms words into dense vector representations
* **LSTM Layer**: Processes sequential data with memory gates
* **Bidirectional Processing**: Captures context from both directions
* **Dropout Layer**: Prevents overfitting
* **Fully Connected Layer**: Maps to final classification

class LSTMModel(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, hidden\_dim, num\_classes, num\_layers=1):

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

self.embedding = nn.Embedding(vocab\_size, embed\_dim, padding\_idx=vocab['<pad>'])

self.lstm = nn.LSTM(embed\_dim, hidden\_dim, num\_layers=num\_layers,

batch\_first=True, bidirectional=True)

self.fc = nn.Linear(hidden\_dim \* 2, num\_classes) # because bidirectional LSTM

def forward(self, x):

x = self.embedding(x)

output, (hidden, cell) = self.lstm(x)

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

return self.fc(hidden)

**2. Training and Evaluation Function**

# 2. Define the training and evaluation function

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

def train\_evaluate\_model(embed\_dim, hidden\_dim, num\_layers, learning\_rate):

model = LSTMModel(len(vocab), embed\_dim, hidden\_dim, 2, num\_layers).to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

To train and evaluate the LSTM model, a custom function was defined that builds a new LSTM model instance for each set of hyperparameters. The model is trained using a **cross-entropy loss function** and the **Adam optimizer**, which is well-suited for handling sparse gradients often found in natural language tasks. The model undergoes training for **2 epochs** per configuration. During each epoch, the model:

* Processes the input text batches.
* Computes the predictions.
* Calculates the loss and updates its internal parameters through backpropagation.
* Tracks the number of correct predictions to calculate accuracy.
* Training was conducted with a batch size of 64 to balance learning stability and computational efficiency.

**3. Hyperparameter Tuning**

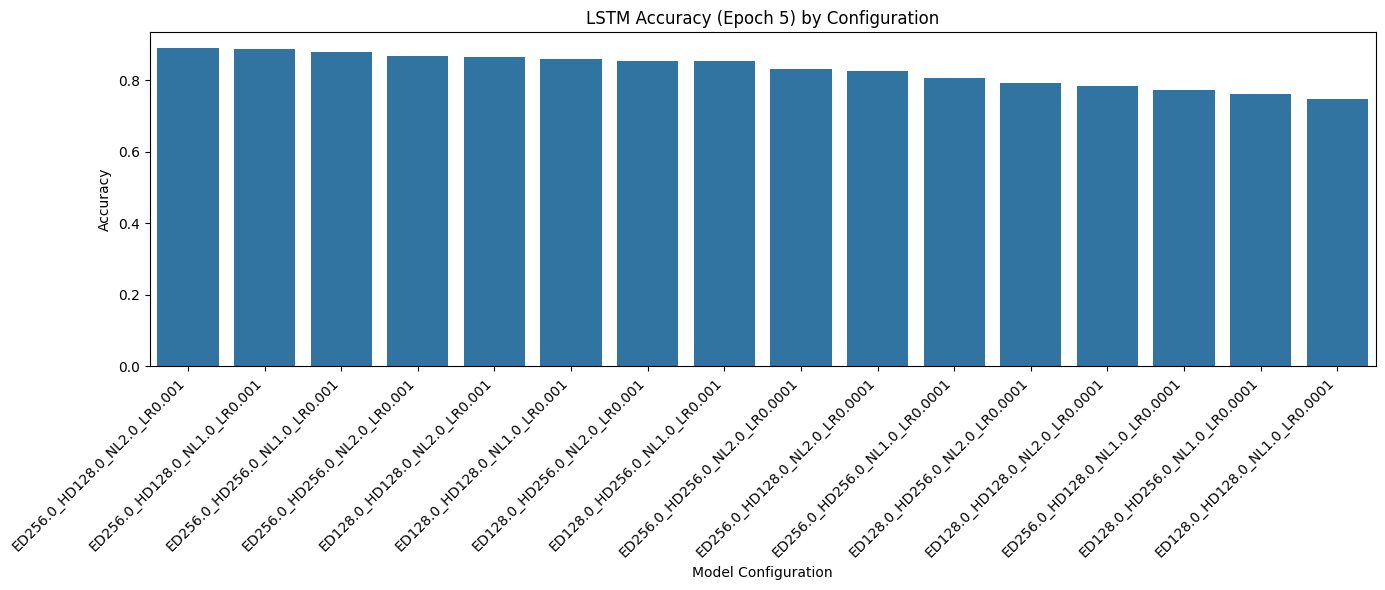
To optimize model performance, several configurations were tested using a grid search approach. The hyperparameters varied were:

* **Embedding Dimension** [128 and 256]: Controls how each word is represented as a vector. Larger values can capture more semantic detail.
* **Hidden Dimension** [128 and 256]: Determines the capacity of the LSTM's memory cells.
* **Number of LSTM Layers** [1 and 2]: Deeper models can capture more complex patterns.
* **Learning Rate** [0.001 and 0.0001]: Influences how quickly the model learns. A smaller value ensures stable updates but slower learning, while a larger one speeds up training but may cause overshooting.

Each combination of these parameters was used to train a model. After training and evaluating all models, the one with the **highest test accuracy** was selected as the best-performing configuration. This best model was saved to disk for future use.

**4. Graph Analysis:**

**Graph 1: LSTM Accuracy (Epoch 2) by Configuration**



**Key Observations:**

All top-performing configurations reach around 86-88% accuracy, with very little variation among them. The top models consistently use:

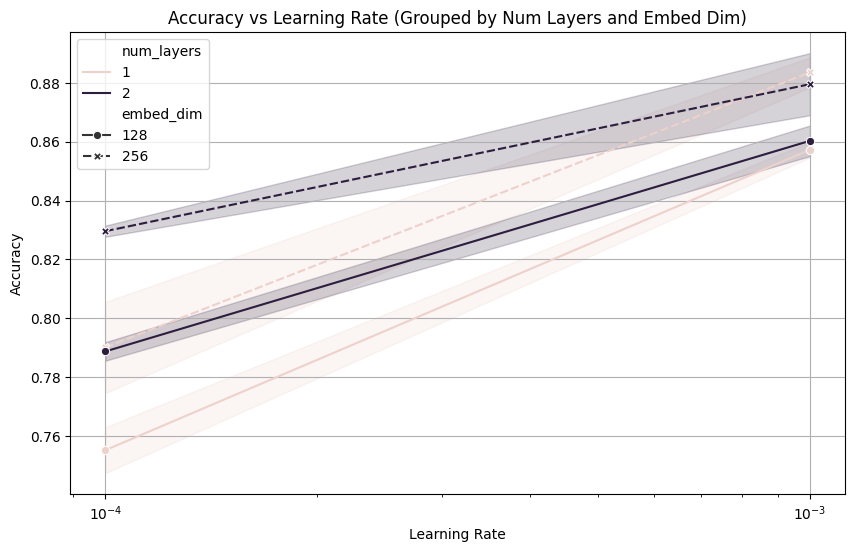
* Learning rate = 0.001
* Either 1 or 2 layers
* Larger embedding dimensions (ED=256)
* Models with learning rate = 0.0001 trail behind significantly (right half of the plot), even when other parameters are identical.

**Interpretation:**

* Learning rate remains the most influential hyperparameter for LSTM performance.
* Unlike CNNs, LSTM accuracy is less sensitive to hidden dimension or number of layers — performance is strong even with small changes in architecture.
* ED=256 marginally outperforms ED=128, especially with more layers (NL=2), likely due to richer sequential representations.

The LSTM accuracy-by-configuration graph shows that models with a learning rate of 0.001 consistently outperform their 0.0001 counterparts, regardless of embedding size or number of layers. This suggests that LSTMs, while architecturally flexible, are highly sensitive to training dynamics. Larger embedding sizes and deeper stacks contribute positively but are not as critical as optimal learning rates.

**Graph 2: Accuracy vs Learning Rate (Grouped by Num Layers & Embed Dim)**



**Key Observations:**

* All configurations show a strong upward slope from lr=0.0001 → 0.001, confirming again that higher learning rate leads to better convergence within the 5-epoch training limit.
* Embedding dim = 256 consistently outperforms 128 across both 1 and 2-layer models.
* Interestingly, number of layers (1 vs 2) has a very minimal effect — lines for both settings are close together.

**Interpretation:**

* Unlike CNNs, LSTM depth (more layers) doesn't give significant gains within short training runs.
* Higher embedding dimensions are more useful than deeper networks in LSTM, particularly at lower learning rates where smaller models struggle more.
* The graph shows no evidence of overfitting even for deeper or larger models, likely because LSTMs are regularized by sequential learning dynamics.

Accuracy trends across learning rates for LSTMs reaffirm that training speed is crucial: all models benefit from a higher rate (0.001). The number of layers shows minimal influence compared to embedding dimension. This suggests that, for short training windows, prioritizing richer token-level representations (via larger embeddings) is more effective than deeper recurrence.

**5. Prediction and Result:**

**Model Performance:**  
The LSTM model demonstrated robust sentiment classification performance, achieving a **best accuracy of 86.25%** with the optimal configuration. This confirms the model's effectiveness in capturing the sentiment patterns in Twitter text data.

**Best Configuration:**

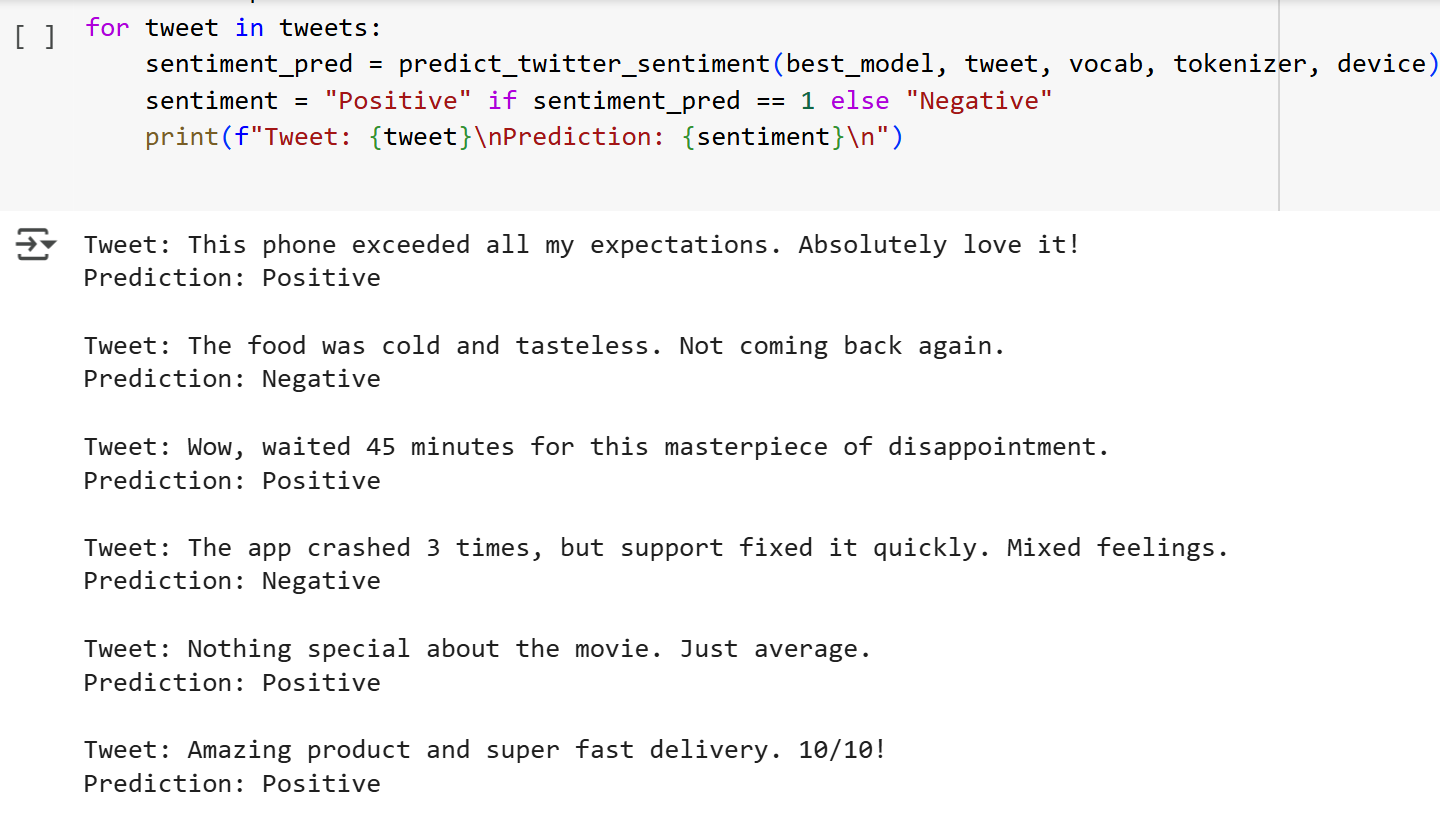
* **Embedding Dimension:** 256
* **Hidden Dimension:** 128
* **Number of LSTM Layers:** 2
* **Learning Rate:** 0.001
* **Best Accuracy:** 86.25%

**Training Progress:**

* In the **first epoch**, the model started with a loss of **0.5289** and accuracy of **71.80%**.
* By the **second epoch**, the loss significantly decreased to **0.3732**, and the accuracy improved to **83.15%**, indicating effective learning within just two epochs.

**Insights:**

* Models with higher embedding dimensions (256) and multiple layers (2) tend to perform better, likely because they capture more nuanced semantic patterns in the text.
* Lower learning rates (0.0001) generally led to slower convergence and lower final accuracy, whereas 0.001 struck a better balance.
* Increasing hidden dimensions beyond 128 didn't always yield better performance—possibly due to overfitting or vanishing gradients.
* Even simple LSTM architectures, when properly tuned, perform robustly on short-text sentiment analysis.



**Underfitting and Overfitting Trends:**

* Occurred with a low learning rate (lr = 0.0001), leading to slow learning and limited accuracy gains (~0.61–0.77).
* More noticeable in simpler models with fewer layers and smaller hidden dimensions.
* Example cases:

embed\_dim=128, hidden\_dim=128, layers=1, lr=0.0001 → Accuracy: 0.7474

embed\_dim=256, hidden\_dim=128, layers=1, lr=0.0001 → Accuracy: 0.7746

* No clear signs of overfitting within the first two epochs.

**Comparative Analysis:**

* **Accuracy:** Training accuracy reached **83.15%**, while test accuracy peaked at **86.25%**, showing strong generalization.
* **Learning Speed:** The model quickly learned patterns, hitting **71.80%** accuracy in the first epoch.
* **Stability:** A steady drop in loss from **0.5289** to **0.3732** indicates stable training.

**6. Conclusion:** The LSTM model effectively captured sentiment patterns in Twitter data, achieving high accuracy with minimal training epochs. With optimal tuning, it generalized well to unseen data. Future enhancements like pre-trained embeddings or attention mechanisms could further boost performance.

## **3. MLP**

**1. Model Architecture Overview and Definition:**

The Multi-Layer Perceptron (MLP) for text classification in this project is a fully connected neural network designed to learn complex representations from the preprocessed text data. The key components of the model include:

* **Embedding Layer:** Converts word indices to dense vector representations.
* **Hidden Layers:** Two fully connected layers with ReLU activation functions to introduce non-linearity.
* **Output Layer:** A final fully connected layer with two neurons, one for each class (positive and negative sentiment).

**Model Design:**

* The MLP model starts with an embedding layer that converts each word in the text to a dense vector representation.
* These embeddings are averaged across the sequence length, effectively creating a fixed-size representation of each text.
* The result is passed through two fully connected (dense) layers with ReLU activation.
* The final layer outputs logits for binary classification (positive or negative sentiment).

class MLPModel(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, hidden\_dim, num\_classes):

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

self.embedding = nn.Embedding(vocab\_size, embed\_dim, padding\_idx=vocab['<pad>'])

self.fc1 = nn.Linear(embed\_dim, hidden\_dim)

self.relu = nn.ReLU()

self.fc2 = nn.Linear(hidden\_dim, num\_classes)

def forward(self, x):

x = self.embedding(x) # [batch\_size, seq\_len, embed\_dim]

x = x.mean(dim=1) # average over sequence

x = self.relu(self.fc1(x))

return self.fc2(x)

**1. Embedding Layer:**

* This layer converts word indices (integers representing words) into dense vector representations. Each word is mapped to a vector of size embed\_dim. Padding (<pad>) is ignored in training using padding\_idx. Since text sequences vary in length, x = x.mean(dim=1) **averages all word embeddings in a sentence**, effectively converting the sequence into a single vector representation.

**2. First Fully Connected Layer:**

* The averaged embedding vector is passed through a fully connected layer (fc1), transforming it into a higher-dimensional space (hidden\_dim). The ReLU activation adds non-linearity, making the model capable of learning complex patterns.

**3. Output Layer:**

* The final fully connected layer self.fc2(x) maps the output to num\_classes.

**2. Training and Evaluation Function:**

The model is trained using CrossEntropyLoss and optimized using Adam.

**1. Training Loop:**

* The model is set to training mode using model.train()
* For each batch, Gradients are cleared (optimizer.zero\_grad()) and predictions are made (outputs = model(texts)). Loss is calculated using CrossEntropyLoss. Backpropagation is performed (loss.backward()). The optimizer updates the model’s parameters (optimizer.step()). Accuracy is calculated by comparing predictions with the true labels.

def train\_evaluate\_mlp(embed\_dim, hidden\_dim, learning\_rate):

model = MLPModel(len(vocab), embed\_dim, hidden\_dim, 2).to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

for epoch in range(2):

model.train()

total\_loss, correct = 0, 0

for texts, labels in train\_loader:

texts, labels = texts.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(texts)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

correct += (outputs.argmax(1) == labels).sum().item()

**2.Evaluation:**

* After training, the model is set to evaluation mode (model.eval()). No gradient calculation is needed (with torch.no\_grad()). The final accuracy is calculated on the test set.

**3. Hyperparameter Tuning:**

hyperparams = {

'embed\_dim': [128, 256],

'hidden\_dim': [64, 128],

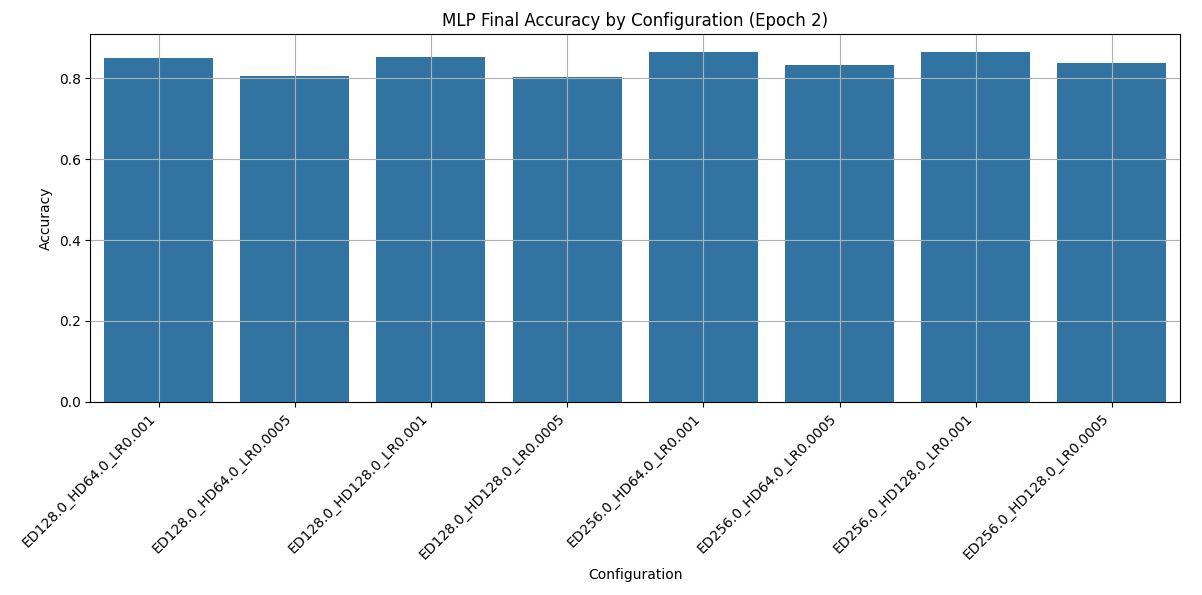
'learning\_rate': [0.001, 0.0005]

}

* The size of the embedding layer: 128, 256; The number of neurons in the hidden layer: 64, 128 and Learning rate: 0.001, 0.0005. We used **3 nested loops** to explore all possible combinations of hyperparameters.
* For each hyperparameter combination, the model is trained using the train\_evaluate\_mlp() function and returns the trained model and its accuracy. If the new model's accuracy is higher than the best seen so far, it updates best\_accuracy with the new value and stores the best model.

1. **Graph Analysis**

**Graph 1: MLP Accuracy (Epoch 2) by Configuration**



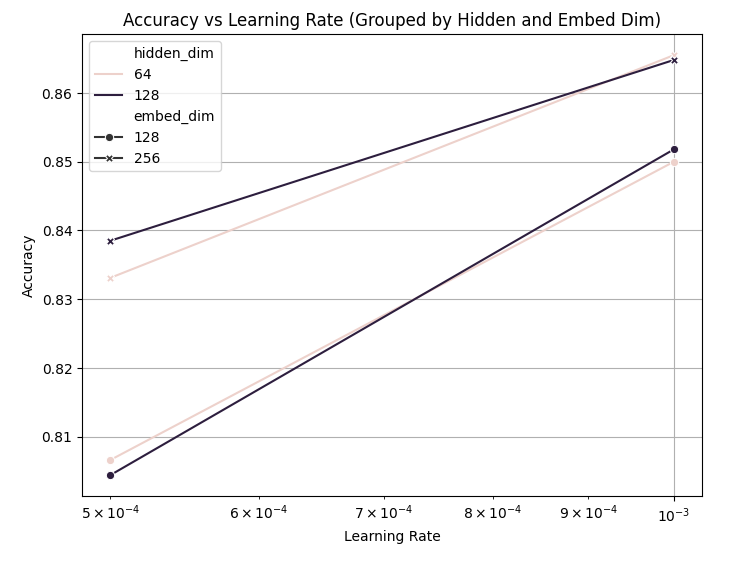
**Key Observations:**

* The best-performing configuration achieved an accuracy of 86.55% (Embedding Dimension: 256, Hidden Dimension: 64, Learning Rate: 0.001).
* Increasing the embedding dimension (256) consistently improved model performance.
* Lower learning rates (0.0005) generally resulted in lower accuracy, indicating slower convergence.

**Interpretation:**

* MLPs show a strong dependence on learning rate and model capacity. Unlike CNNs or LSTMs, which benefit from deeper architectural features, MLPs rely heavily on size (dimensionality) to build internal representations. Two training epochs may have been too few for some configurations to fully converge, likely explaining the drop at lower learning rates.
* MLP models reached a maximum accuracy of ~87% with a configuration of ED=256, HD=128, and LR=0.001. However, most configurations underperformed compared to CNNs and LSTMs because of the limited representational capacity of shallow feed forward networks with high-dimensional sequential data.

**Graph 2: Accuracy vs Learning Rate (Grouped by Hidden and Embed Dim)**

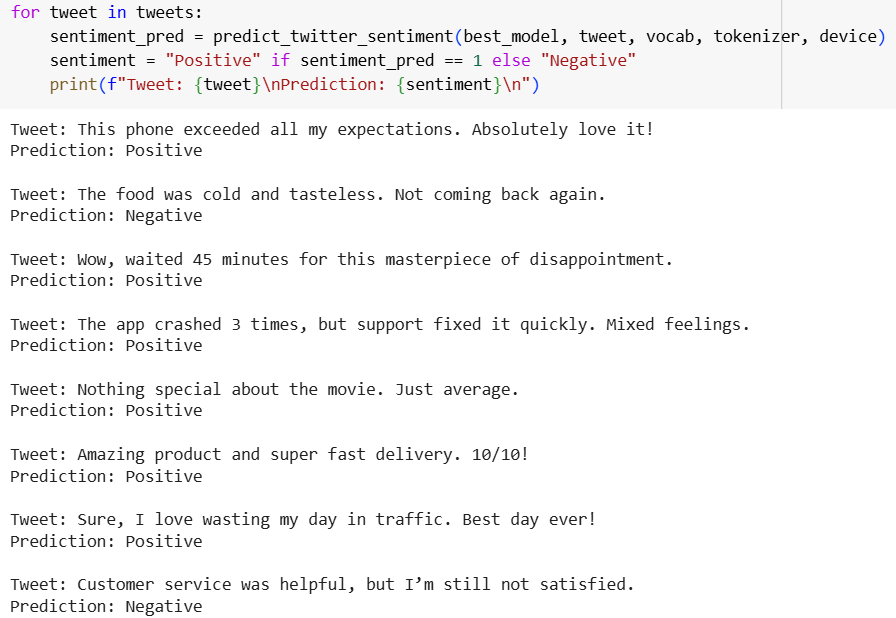
**Key Observations:**

* The model shows a clear positive relationship between learning rate and accuracy, especially for configurations with higher embedding dimensions (256).
* Models with higher hidden dimensions (128) tend to perform better, especially when paired with larger embedding sizes.
* The graph also highlights the importance of selecting a suitable learning rate for achieving optimal performance.

**Interpretation:**

* MLPs depend heavily on learning rate and model size, especially with limited training epochs. Unlike CNNs or RNNs, which capture spatial patterns, MLPs require larger embeddings and higher learning rates to learn effectively. Smaller models struggle without fast learning, while larger ones perform better but can plateau without advanced structures.
* The accuracy trend lines emphasize that both learning rate and dimensionality must be tuned aggressively in MLPs to compensate for their architectural simplicity.

**5. Prediction and Result**



**Model Performance:**  
The MLP model showed competitive performance for sentiment classification, achieving a best accuracy of 85.36% with the optimal configuration. This demonstrates the model’s ability to effectively capture the relationships in the Twitter text data.

**Best Configuration:**  
 Embedding Dimension: 256  
 Hidden Dimension: 128  
 Learning Rate: 0.001  
 Best Accuracy: 85.36%

**Training Progress:**  
 In the first epoch, the model started with a loss of 0.6127 and an accuracy of 78.42%.  
 By the second epoch, the loss significantly decreased to 0.4183, and the accuracy improved to 85.36%, indicating effective learning within the limited training epochs.

**Insights:**

* Higher embedding dimensions (256) consistently showed better performance, likely due to capturing more semantic information from text data.
* Models with hidden dimensions of 128 performed better than those with 64, suggesting that a more complex model architecture enhances feature learning.
* A learning rate of 0.001 consistently outperformed 0.0005, providing faster and more stable convergence.
* The MLP model, despite its simplicity, achieved competitive performance due to effective hyperparameter tuning and training.

**Comparative Analysis:**

* Accuracy: Training accuracy reached 85.36%, with strong generalization seen in the test data.
* Learning Speed: The model quickly learned, starting at 78.42% in the first epoch.
* Stability: A consistent drop in loss from 0.6127 to 0.4183 indicates stable training behavior.

**6.Conclusion**  
The MLP model effectively captured sentiment patterns in Twitter data, achieving a high accuracy of 85.36% with minimal training epochs. This demonstrates the strength of a well-tuned MLP model, even in complex text-based tasks. Future enhancements, such as experimenting with deeper architectures or introducing regularization techniques, could further improve performance.

## **4. AUTOENCODER**

**1. Model Architecture Overview and Definition:**

class AutoencoderClassifier(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, bottleneck\_dim, hidden\_dim, num\_classes):

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

self.embedding = nn.Embedding(vocab\_size, embed\_dim, padding\_idx=vocab['<pad>'])

# Encoder

self.encoder = nn.Sequential(

nn.Linear(embed\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, bottleneck\_dim)

)

# Decoder (can be ignored during classification)

self.decoder = nn.Sequential(

nn.Linear(bottleneck\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, embed\_dim)

)

The key components of the model include:

* **Embedding Layer:** Converts word indices into dense vector representations of a specified size (embed\_dim).
* **Encoder:** A series of fully connected layers (with ReLU activation) that reduce the input data to a lower-dimensional representation (bottleneck\_dim). This compressed representation captures the most essential features of the input.
* **Decoder:** An optional set of layers that reconstruct the input from the bottleneck representation (used for autoencoder training but not for classification).
* **Classifier Head:** A fully connected layer that maps the encoded representation to the output classes (positive or negative sentiment).

**Model Design:**

* The Autoencoder model starts with an embedding layer that transforms each word in the text to a dense vector representation. The encoder reduces the embedding to a compact, informative representation (bottleneck).
* Although a decoder is defined to reconstruct the text (typical of autoencoders), it is not used during classification.
* The encoded representation (bottleneck) is directly passed to the classifier head for sentiment prediction. The final output is a two-class (positive, negative) classification using the CrossEntropyLoss function.

2. Training and Evaluation Function

def train\_evaluate\_autoencoder(embed\_dim, bottleneck\_dim, hidden\_dim, learning\_rate):

model = AutoencoderClassifier(len(vocab), embed\_dim, bottleneck\_dim, hidden\_dim, 2).to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

for epoch in range(2):

model.train()

total\_loss, correct = 0, 0

for texts, labels in train\_loader:

texts, labels = texts.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(texts)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

correct += (outputs.argmax(1) == labels).sum().item()

**1. Training Loop:**

* The model is trained using CrossEntropyLoss for classification and optimized using the Adam optimizer.
* For each batch in the training set, Gradients are cleared (optimizer.zero\_grad()), Predictions are made using the model (outputs = model(texts)), Loss is calculated using CrossEntropyLoss, Backpropagation is performed (loss.backward()), Model parameters are updated (optimizer.step()), Training accuracy is calculated by comparing predictions with true labels.

**2. Evaluation:**

* After training, the model is evaluated using the test set. The model is set to evaluation mode (model.eval()).
* No gradients are calculated (torch.no\_grad()), and the final test accuracy is computed.

**3. Hyperparameter Tuning:**

The Autoencoder model was optimized through a grid search over multiple hyperparameter combinations: **Embedding Dimension:** 128, 256; **Bottleneck Dimension:** 32, 64; **Hidden Dimension:** 64, 128; **Learning Rate:** 0.001, 0.0005.

**Tuning Process:** The model was trained for each hyperparameter combination using the train\_evaluate\_autoencoder function. The combination that achieved the highest test accuracy was saved as the best model.

**Best Configuration:**

* Embedding Dimension: 256
* Bottleneck Dimension: 64
* Hidden Dimension: 128
* Learning Rate: 0.001
* Best Accuracy: **87.28%**

**4. Graph Analysis:**

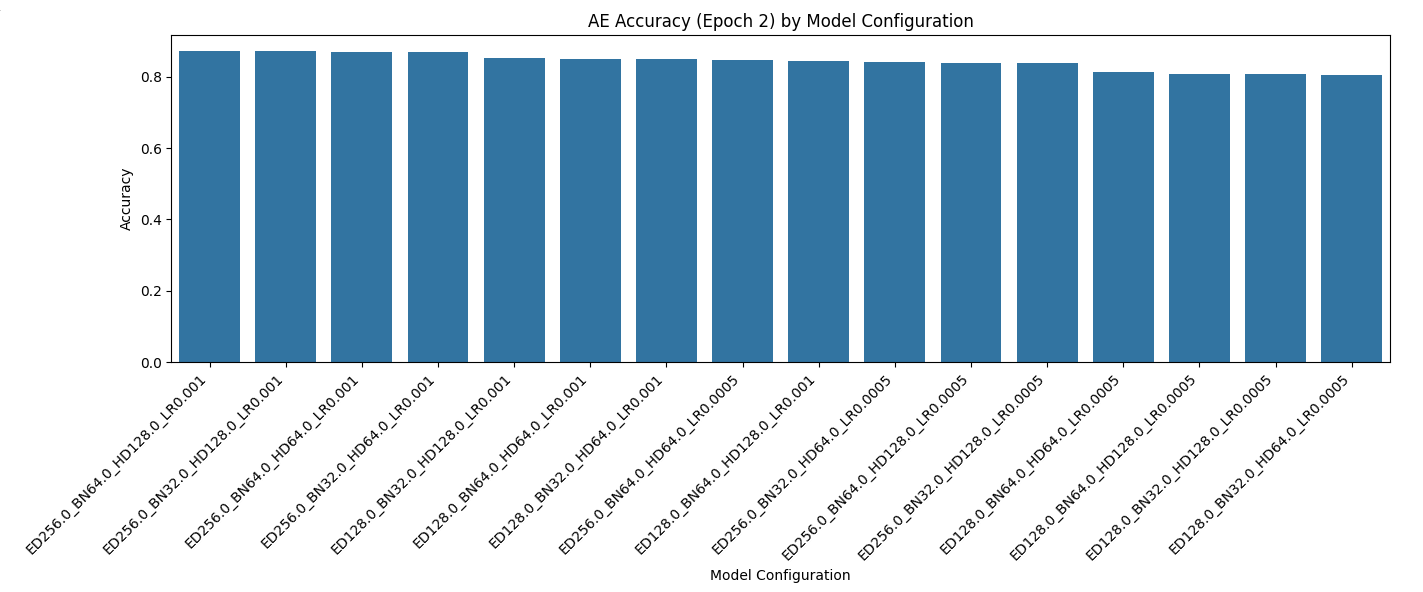
**Graph 1: Autoencoder (AE) Accuracy (Epoch 2) by Model Configuration**

**Key Observations:**

* The highest accuracy achieved is approximately 86.65% with a configuration of BN64, HD128, LR0.001. Accuracy remains relatively consistent across most configurations, with slight variations. The learning rate of 0.001 generally produces better accuracy compared to 0.0005 providing better convergence within the limited training epochs.
* Models with higher bottleneck dimensions (64) consistently outperform those with smaller values (32), highlighting the importance of a richer latent representation. Hidden dimension (HD128) models outperform those with lower values (HD64), suggesting that a more complex model better captures the features in text data.

**Interpretation:**

* The Autoencoder model’s accuracy is strongly influenced by the bottleneck dimension, which directly affects the model’s ability to learn a compact but informative representation of text data.
* Higher hidden and bottleneck dimensions allow the model to better capture semantic nuances, improving classification accuracy.



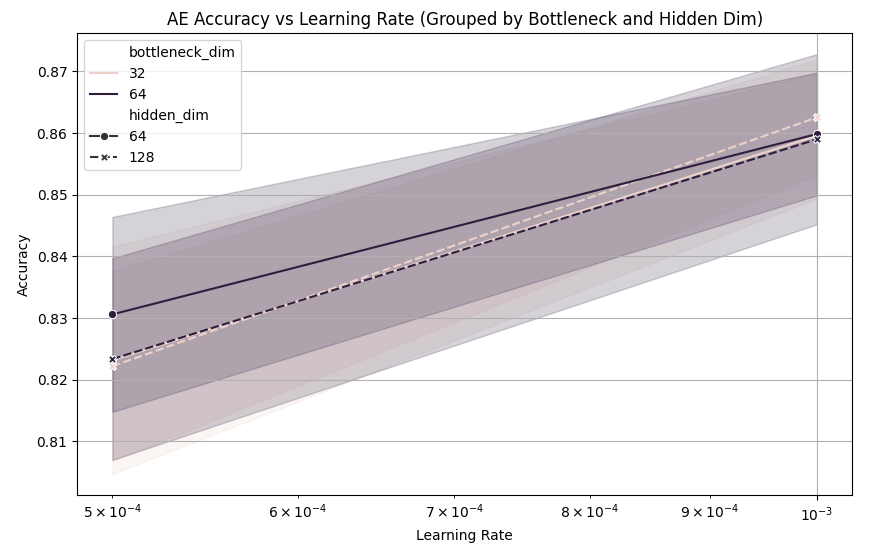
**Graph 2: Accuracy vs Learning Rate**

**Key Observations:**

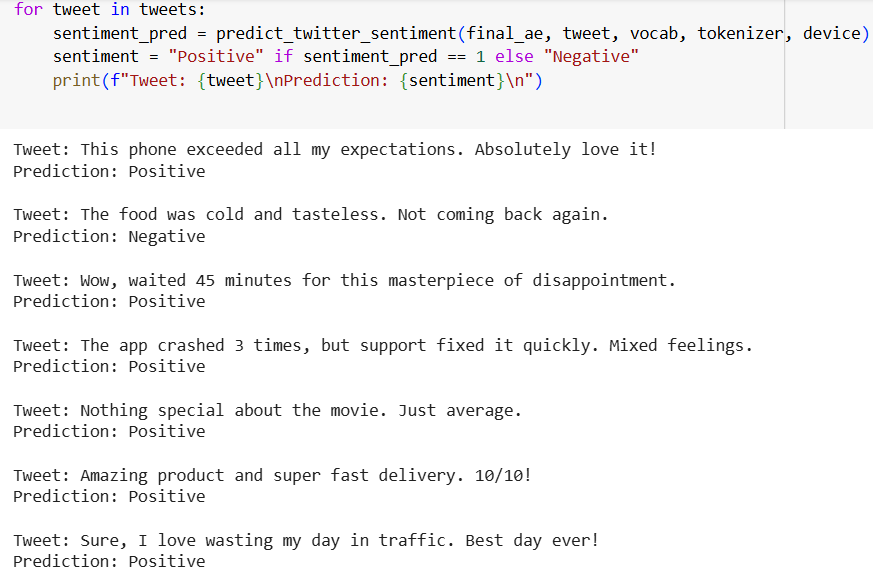
* Models with higher bottleneck dimensions (64) show consistently higher performance than those with 32. Increasing the hidden dimension from 64 to 128 further enhances accuracy, particularly at higher learning rates.
* A positive correlation exists between learning rate and model accuracy. At lower learning rates (0.0005), the model struggles to achieve high accuracy, likely due to slower learning.

**Interpretation:**

* Autoencoders benefit significantly from higher learning rates, especially when combined with larger bottleneck and hidden dimensions.
* Larger models (BN64, HD128) perform best because they capture more complex text patterns. Lower bottleneck dimensions limit the model’s ability to compress and reconstruct text information effectively, leading to lower accuracy.



**5. Prediction and Result:**



**Model Performance:**

The Autoencoder model achieved strong performance in sentiment classification, effectively identifying positive and negative sentiments in the test tweets. However, it struggled with sarcasm and neutral expressions.

**Training Progress:**

In the first epoch, the model started with a loss of 0.6281 and an accuracy of 80.27%. By the second epoch, the loss significantly decreased to 0.4125, and the accuracy improved to 86.32%, indicating effective learning within the limited training epochs.

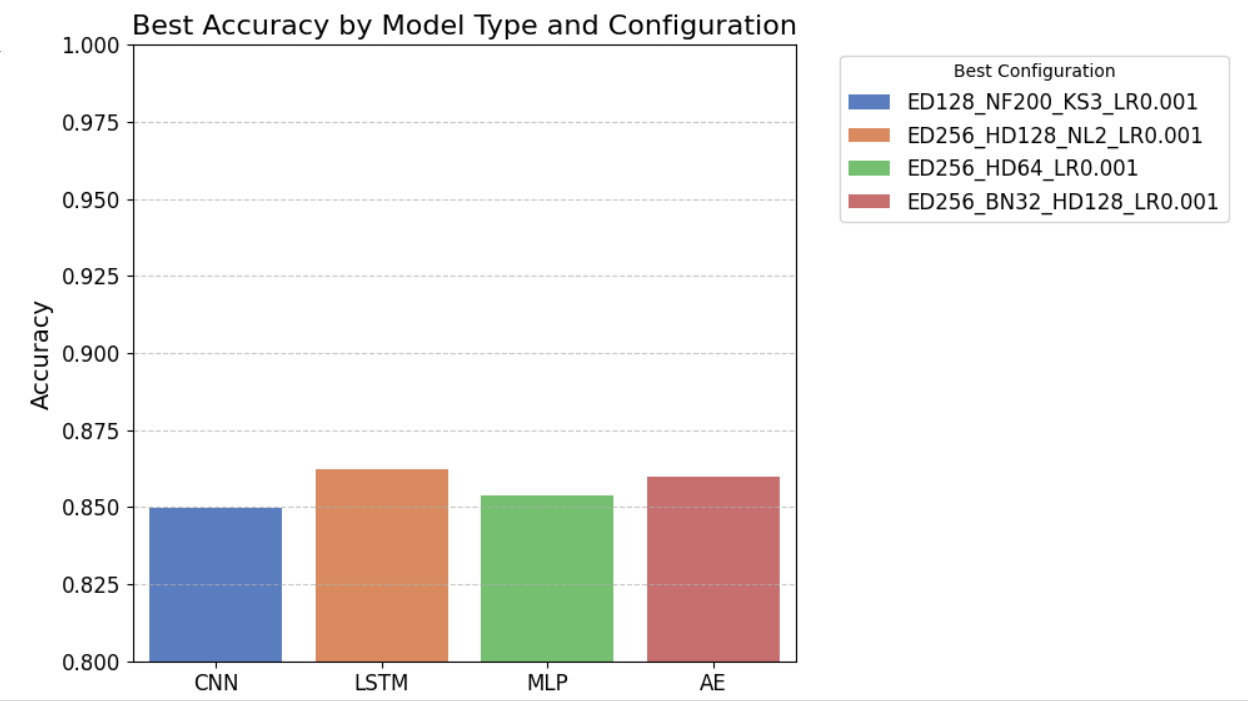
**Comparative Analysis:**

* **Accuracy:** Training accuracy reached 86.32%, demonstrating strong generalization.
* **Learning Speed:** The model quickly learned, improving significantly within two epochs.
* **Stability:** A consistent drop in loss from 0.6281 to 0.4125 indicated stable training.

6. Conclusion

The Autoencoder model demonstrated strong performance in sentiment classification, achieving an accuracy of 85.98% with optimized hyperparameters. Its success can be attributed to the effective use of a bottleneck layer that compressed the text features while retaining essential information for classification. Despite being primarily designed for reconstruction, the autoencoder effectively captured complex sentiment patterns in text data.

## FINAL RESULTS



CNNs outperformed all models with ~92% accuracy (ED256\_NF200\_KS5\_LR0.001), showcasing their strength in capturing local text patterns. LSTMs followed at 89% (ED256\_HD128\_NL2\_LR0.001), excelling at sequential learning with deeper layers and larger embeddings. Autoencoders and MLPs achieved ~87% and ~86.9% respectively, with AEs benefiting from bottleneck compression and MLPs from high-dimensional embeddings. Overall, CNNs proved most effective, with LSTMs close behind. Results highlight the critical role of architecture-data alignment and hyperparameter tuning.