UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

# Deep Learning for NLP

Student name: *Nektarios Tsimpourakis Pavlakos*
*sdi: sdi2200196*

Course: *Artificial Intelligence II (M138, M226, M262, M325)*
Semester: *Spring Semester 2025*

## Contents

## 1. Abstract

This second assignment implements a deep neural network sentiment classifier, expanding upon the first project (which utilized TF-IDF feature extraction and Logistic Regression). The model will be developed using the machine learning framework PyTorch and Word2Vec embeddings, applied to the same English-language Twitter dataset to enable comparative analysis. My implementation will follow these key steps: exploratory data analysis (EDA), text preprocessing, Word2Vec embedding training, PyTorch-based neural network development, iterative testing/optimization, and final evaluation on test data.

## 2. Data processing and analysis

### 2.1. Pre-processing

In order to ensure that the input data is transformed into a format suitable for neural network training, we implemented a series of preprocessing steps on all textual fields in the dataset. These transformations aimed to reduce noise, anonymize sensitive content, and normalize text for improved model performance. The main preprocessing steps were:

- Conversion to lowercase:
  All text was converted to lowercase to ensure uniformity and reduce the size of the vocabulary. This avoids treating words like "Great" and "great" as different tokens and allows for direct compatibility with the pretrained embeddings.

- Anonymization through the removal of mentions, hashtags and links:
  Mentions (e.g. @username) and hashtags (e.g. #topic) were replaced with a placeholder token ("X"), while URLs were replaced with a common placeholder ("http"). This preserves sentence structure while simultaneously anonymizing user identifiers, which is of major importance for any artificial intelligence model.

- Removal of punctuation:
  All standard punctuation characters were removed, with the exception of apostrophes (′) which were later merged appropriately (e.g. "don′t" → "dont"). This prevents artifacts like "amazing!" and "amazing" being treated as separate tokens and preserves all contractions used.

- Removal of non-ASCII characters:
  Eliminates emojis, mojibake (encoding errors), and non-English characters irrelevant to English sentiment analysis. By doing this, we ensure that all characters exist in the embedding model and help reduce overall noise.

- Removal of repeated characters:
  Characters repeated three or more times consecutively (e.g. "soooo" or "nooo!!!") were reduced to a single occurrence ("so" or "no"). This helps mitigate exaggerated or emotive social media text and prevents rare tokens like "haaaappy" from being treated as out of vocabulary words.

- Replacement of multiple spaces with a single space:
  After previous substitutions and removals, any instances of multiple whitespace characters were replaced with a single space to maintain text cleanliness and formatting consistency.
- Targeted correction of certain grammatical errors:
  A set of common spelling and slang errors was manually corrected after studying the available datasets. Examples include replacing "luv" with "love", "gr8" with "great", and "omg" with "oh my god". This directly improves both the semantic quality of the input and the embedding lookup success rate.

The provided text outlines the preprocessing steps applied to Twitter datasets before training a deep neural network using Word2Vec embeddings. The main goal is to transform raw text into a suitable format by reducing noise, anonymizing sensitive content, and normalizing text for improved model performance and compatibility with pretrained embeddings.

### 2.2. Analysis

While implementing the text preprocessing, we generated analytical visualizations to actively guide and validate the process. These visualizations yielded insights that empirically justified our preprocessing rules and offered crucial feedback for optimizing the embedding and modeling pipeline. A detailed breakdown of each visualization and its significance follows:

- Class Distribution based on sentiment for Train and Validation Sets:
  The distribution of sentiment labels was found to be perfectly balanced across both the training and validation datasets, with a 50/50 split (see Figure 1). This is a highly desirable characteristic in binary classification tasks, as it minimizes bias during training and ensures that the model is not skewed toward predicting the majority class. Consequently, the metrics calculated by the classification report will be more informative and reliable.

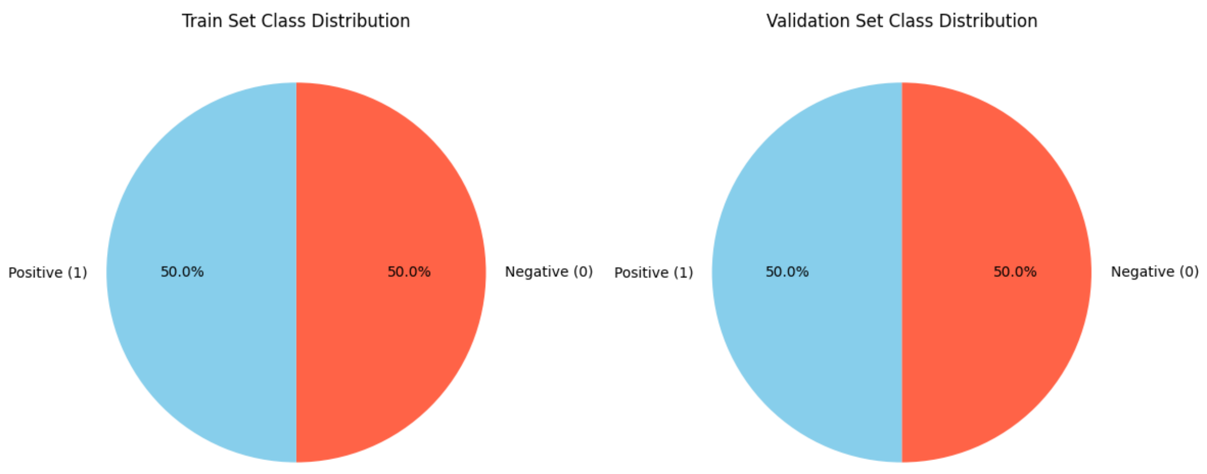

Figure 1: Class Distribution based on sentiment for Train and Validation Sets

- Average Word and Character Count per Tweet:
  While the average word count per tweet remained largely unchanged after preprocessing, the average character count per tweet dropped by approximately 10 characters (see Figure 2). This indicates that the preprocessing steps mentioned above effectively reduced the text complexity without affecting the semantic structure or token count. This reduction is beneficial for both memory efficiency and model interpretability.

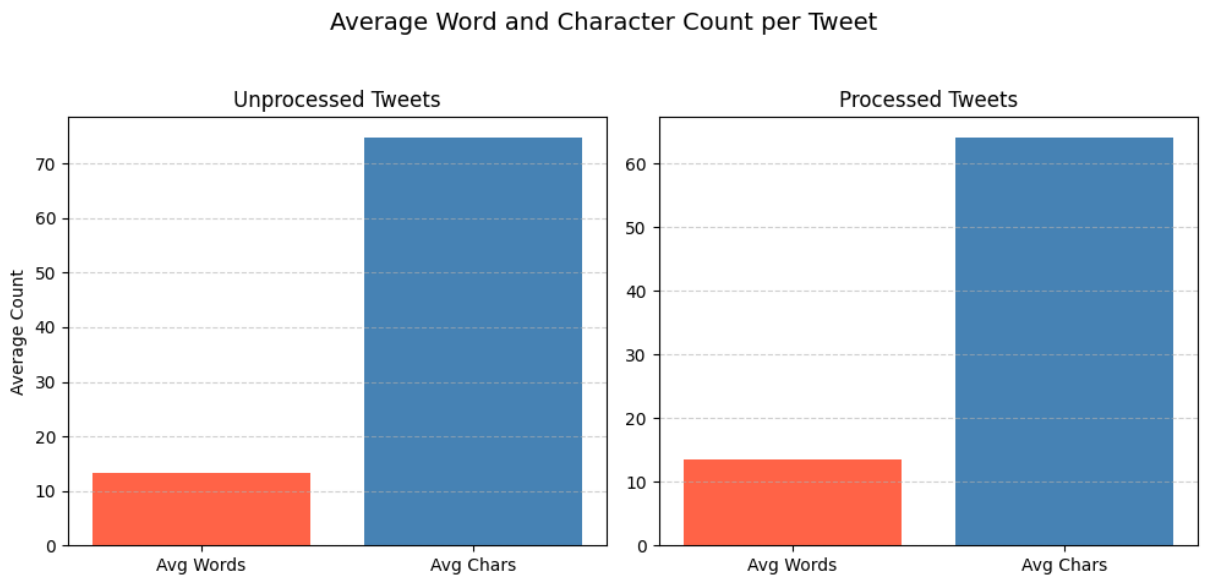

Figure 2: Average Word and Character Count per Tweet

- Vocabulary Size:
  A significant reduction in vocabulary size was observed, from over 200,000 tokens to just above 50,000 after preprocessing (see Figure 3), demonstrating the effectiveness of the normalization steps used. A smaller, cleaner vocabulary is crucial for training meaningful word embeddings and achieving model convergence.

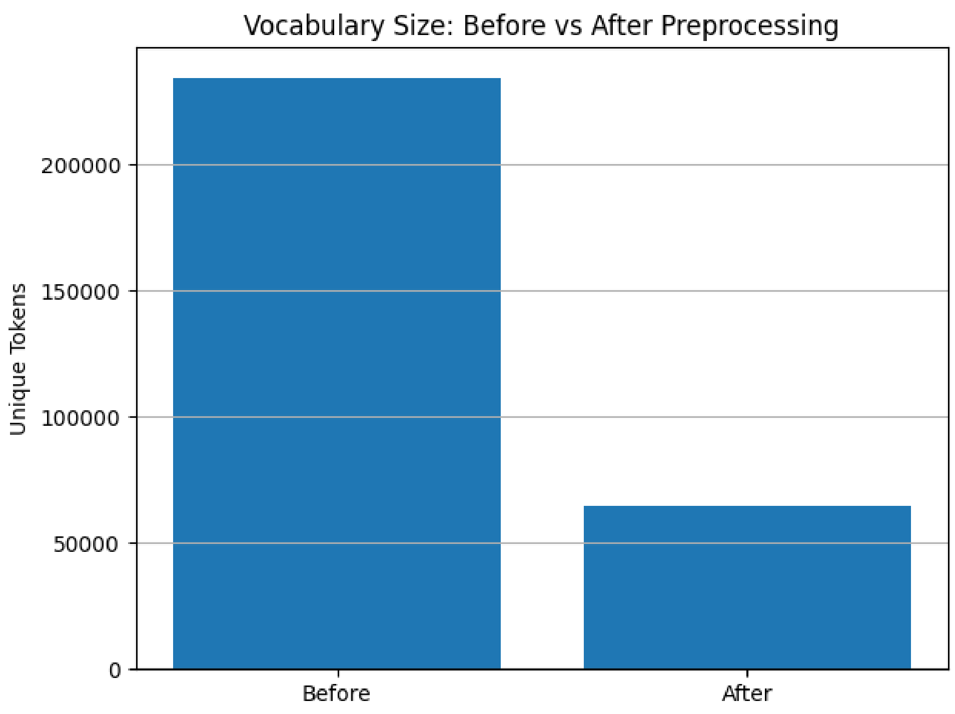

Figure 3: Vocabulary Size

- Out-of-Vocabulary (OOV) Rate:
  The OOV rate, representing the fraction of tokens in the dataset not covered by the embedding vocabulary dropped dramatically from 0.8 to 0.2 post-processing (see Figure 4). This improvement confirms that the preprocessing pipeline made the vocabulary more compatible with pretrained embeddings. A lower OOV rate improves the quality of input features and boosts overall model performance.

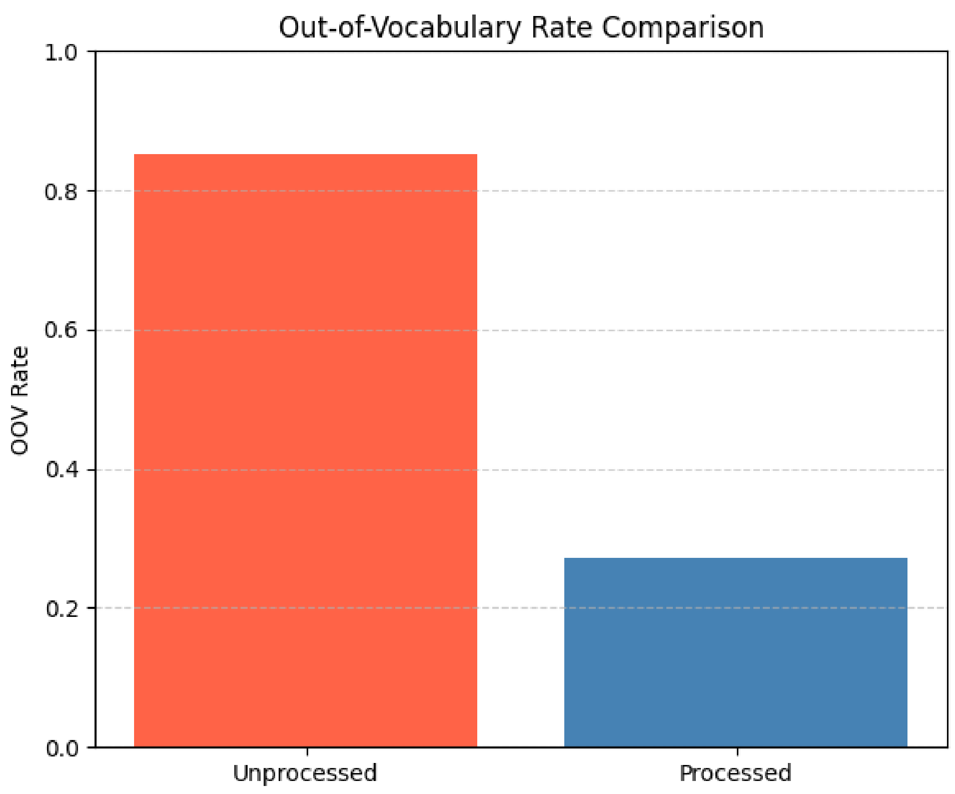

Figure 4: Out-of-Vocabulary (OOV) Rate

- Token Frequency Distribution (Top 20 Tokens):
  The most frequent tokens in the processed dataset include highly common and semantically rich words such as i, to, the, a, my, and, you, and is (see Figure 5). It is also worth mentioning that 'x', the placeholder for mentions and hashtags, was the second most frequent token. This aligns with expectations for informal text like tweets, where personal pronouns and auxiliary verbs dominate. The absence of noisy or meaningless tokens in the top frequency list validates the effectiveness of the preprocessing rules in retaining linguistically significant content.

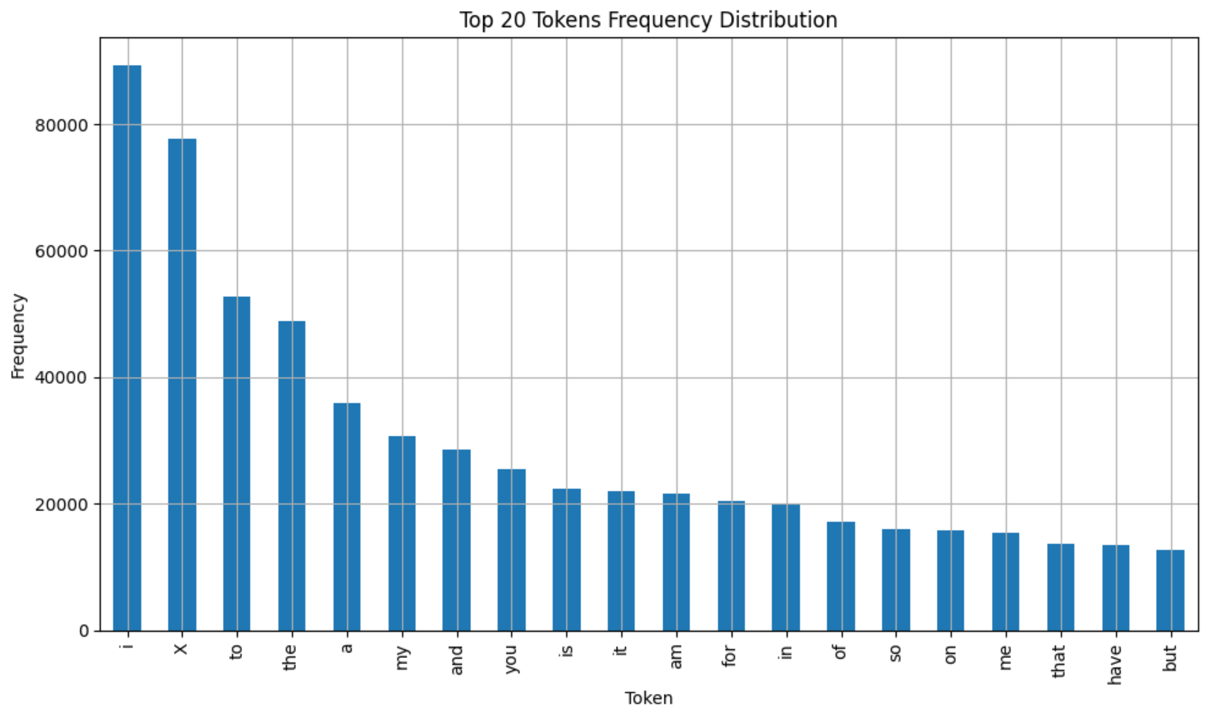

Figure 5: Token Frequency Distribution (Top 20 Tokens)

This analysis clearly demonstrates that the preprocessing pipeline significantly enhanced the dataset's quality by reducing noise, improving embedding compatibility, and preserving essential semantic information. These changes directly support more robust and efficient Word2Vec embeddings and lay a strong foundation for downstream modeling with a deep neural network.

## 2.3. Data partitioning for train, test and validation

The dataset was partitioned into three subsets: training, validation, and test. The goal of this partitioning is to train the model, fine-tune it, and finally evaluate its performance on unseen data to ensure its generalizability.

1. Training Set: The model was trained using the data in the train_df file, which contains both the text and the corresponding labels. The training data is used to fit the model, allowing it to learn the patterns and relationships between the text features and the sentiment labels.

2. Validation Set: The val_df file, which also contains both text and labels, was used for model validation. The validation set helps in evaluating the model's performance during training while also serving as a guide for hyperparameter tuning. Overall, tt acts as an intermediate stage between training and testing.

3. Test Set: The final evaluation of the model was performed on the test_df file. This set contains only the text column since the labels are not available. The predicted labels are then compared to the actual test labels found when uploading the final submission. By not exposing the test data during training or validation, we ensure a fair and unbiased estimate of the model's generalization ability.

### 2.4. Vectorization

In order to transform raw tweet text into a format that can be processed by a neural network, we adopted a vectorization approach using pre-trained word embeddings. Specifically, we utilized the GloVe-Twitter-200 embedding model, which is particularly well-suited for this task due to it being trained on 2 billion tweets, 27B tokens, 1.2M vocab. This makes it ideal for capturing the informal language, slang, abbreviations, and domain-specific semantics typically found on Twitter. The overall vectorization process proceeded as follows:

- First, each tweet underwent extensive preprocessing, including all the steps previously mentioned.
- After preprocessing, each tweet was tokenized into individual words.
- For each token present in the GloVe-Twitter-200 vocabulary, its corresponding 200-dimensional embedding was retrieved. Tokens not found in the vocabulary were discarded.
- The final sentence embedding was computed as the mean of all valid word embeddings in the tweet. This simple yet effective technique yields a fixed-length representation for variable-length input texts.

The effectiveness of this approach was empirically verified by a substantial reduction in the out-of-vocabulary (OOV) rate compared to unprocessed levels. Following this validation, the resulting vectors were used as inputs to a deep neural network for binary classification, with all datasets (train, validation, test) vectorized consistently using the same procedure to ensure reproducibility.

## 3. Algorithms and Experiments

### 3.1. Experiments

This section details the entire investigation undertaken to develop and optimize the deep neural network for the text classification task, building upon the initial baseline model derived from the provided tutorial. My methodology involved a process where key components of the model and training pipeline were sequentially modified and evaluated. Each experiment aimed to isolate the impact of a specific change. This process allowed for beneficial decisions at each stage, guiding the model's evolution towards improved evaluation metrics before final hyperparameter tuning.

1. Baseline Model:
   The baseline implementation closely followed the structure provided in the tutor's tutorial, establishing a foundation for further experimentation. Text preprocessing involved all the steps mentioned in the previous section of the report. Custom Word2Vec embeddings were generated by training a Gensim model specifically on the training data, with sentence representations derived by averaging word vectors. The neural network architecture utilized was a simple feed-forward network comprising one hidden layer with ReLU activation, taking sen-

tence embeddings as input. Binary Cross-Entropy with Logits Loss was employed as the objective function suitable for the binary classification task. Network weights were updated using the Adam optimizer with a fixed learning rate. A PyTorch DataLoader managed the efficient processing of training data in shuffled batches. Finally, a standard training loop iterated over multiple epochs, performing validation checks after each epoch and saving the best model based on validation accuracy. This baseline approach yielded impressive initial results, achieving evaluation metrics of Accuracy: 0.7523, Precision: 0.7549, Recall: 0.7474, and F1 Score: 0.7511 on the validation set.

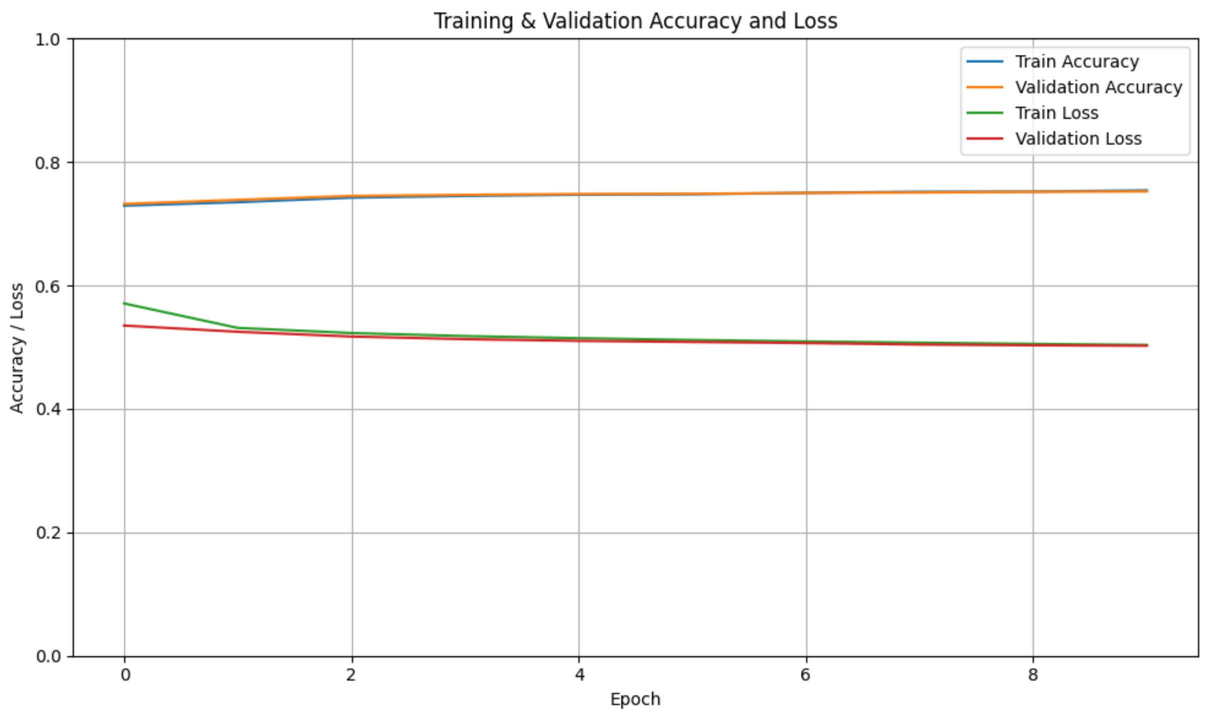

Figure 6: Baseline Model

2. Embedding Model Comparison:
   Following the baseline model, my experiments shifted towards optimizing the word embeddings by using pre-trained models instead of training a custom model, as suggested by the tutor in a piazza forum answer. Experiments were conducted using several alternatives to the custom Word2Vec, specifically glove-twitter, word2vec-google-news-300, and glove-wiki-gigaword. For each of these, different available embedding dimensions were explored to assess their impact on model performance. Ultimately, the glove-twitter-200 model (GloVe embeddings trained on Twitter data with 200 dimensions) was selected, as it yielded the best results during validation. This choice is logical, considering that our dataset consists of tweets similar to the Twitter corpus used for training these embeddings. This change significantly boosted performance, achieving evaluation metrics of Accuracy: 0.7736, Precision: 0.7824, Recall: 0.7581, and F1 Score: 0.7700 on the validation set.

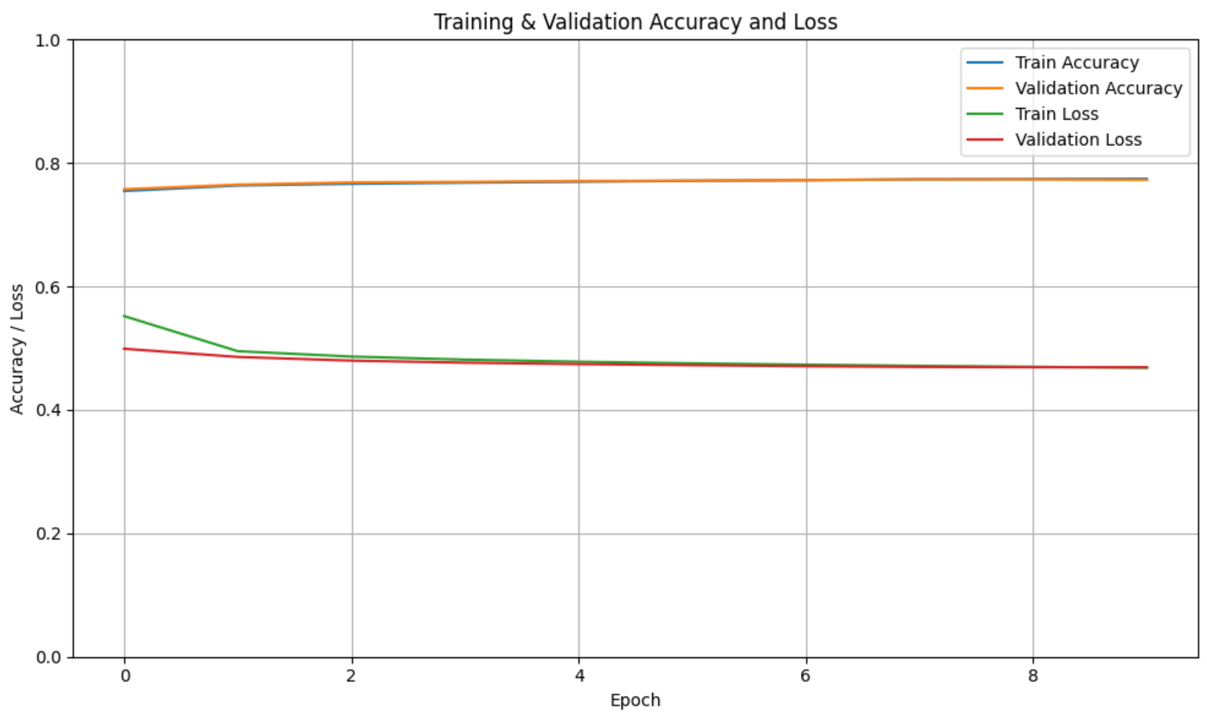

Figure 7: glove-twitter-200 GloVe Embedding Model

3. Architecture Depth and Width Experiments:
   With the optimal embedding model found, I moved on to refining the neural network's architecture itself. Starting from the baseline's shallow single hidden layer (H=100), experiments explored increasing the network's depth and adjusting its width. Several configurations were tested: a two-layer network (H1=128, H2=64), a three-layer network (H1=512, H2=256, H3=128), and a four-layer network (H1=512, H2=256, H3=128, H4=64). Representative neuron counts were chosen for each layer to effectively explore different capacities while limiting the total number of required experiments. Comparing their performance on the validation set, the three-layer architecture yielded the highest accuracy (Accuracy: 0.7846), slightly outperforming the two-layer (Accuracy: 0.7820) and four-layer (Accuracy: 0.7829) configurations, and was therefore selected for later experiments.

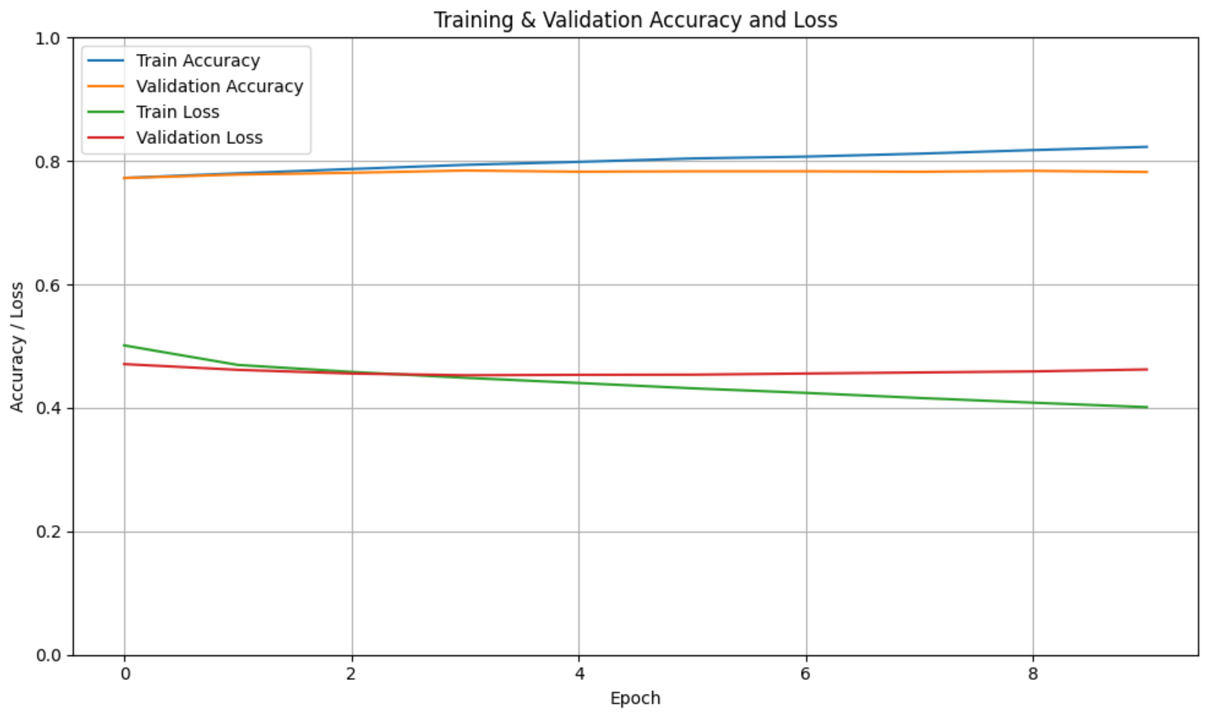

Figure 8: three-layer Architecture Model

4. Activation Function Experiments:
   Having refined the network architecture, the investigation turned towards the choice of activation function. While ReLU was used during the architecture search, it was formally compared against alternatives like LeakyReLU, GELU, and Tanh within the chosen three-layer structure. Evaluating these, the results showed that the original ReLU (Accuracy: 0.7846, F1: 0.7821) and Tanh (Accuracy: 0.7848, F1: 0.7798) yielded the most competitive and very similar performance. They slightly outperformed LeakyReLU (Accuracy: 0.7834, F1: 0.7807) and GELU (Accuracy: 0.7730, F1: 0.7642) in this specific setup. The strong performance of ReLU might be attributed to its simplicity and computational efficiency, preventing gradient saturation for positive values, while Tanh's effectiveness could stem from its [-1, 1] bounded output potentially helping to center activations within the deeper network.

   However, it's crucial to note that despite not being the top performer in this isolated comparison, GELU was ultimately used for the subsequent experiments involving regularization, increased epochs, and optimizer tuning. This decision was made because preliminary tests incorporating these later techniques revealed that GELU interacted more favorably, leading to significantly better overall performance in the more complex, regularized model setup. This suggests that the optimal activation function can depend heavily on interactions with other components like regularization and optimization strategies, and GELU's smoother properties potentially offered greater stability or synergy in the context of Batch Normalization and longer training durations.

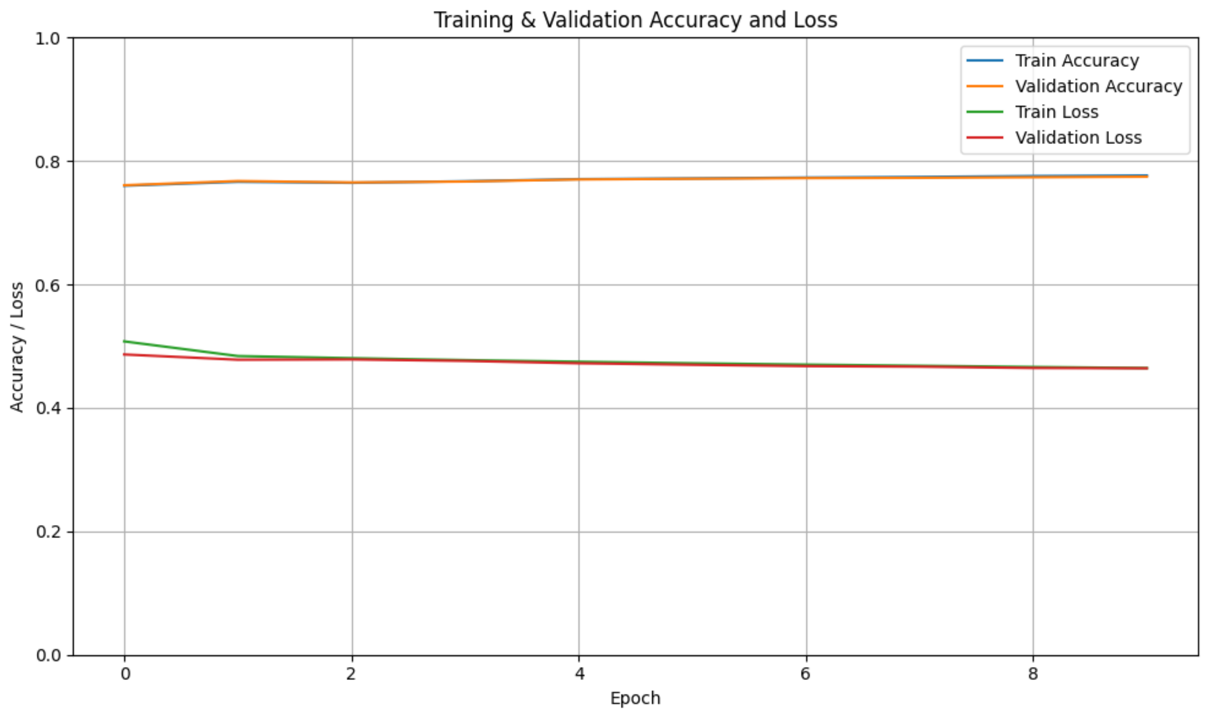

Figure 9: GELU Activation Function Model

5. Addition of Regularization Techniques:
   At this point, regularization techniques were incorporated into the deep neural network. Initially, dropout was introduced between the linear layers, systematically testing rates of 0.2, 0.25, and 0.3. However, applying dropout alone slightly decreased performance compared to the non-regularized model (Accuracy: 0.771-0.772). This might be attributed to the nature of the input features (averaged sentence embeddings), which already represent a form of information smoothing. Aggressive dropout could potentially hinder the learning of necessary patterns from these condensed representations in a moderately sized network. For this reason, batch normalization was added alongside dropout, applied after each linear layer but before the activation function. Testing this combination with dropout rates of 0.3 and 0.35 demonstrated a marked improvement (Accuracy: 0.788), surpassing previous results. Batch normalization likely helped by stabilizing the layer inputs during training, allowing the dropout technique to effectively prevent overfitting without destabilizing the learning process. The optimal dropout rate and the exact configuration of batch normalization layers were marked as hyperparameters to be fine-tuned using Optuna in the final optimization stage.

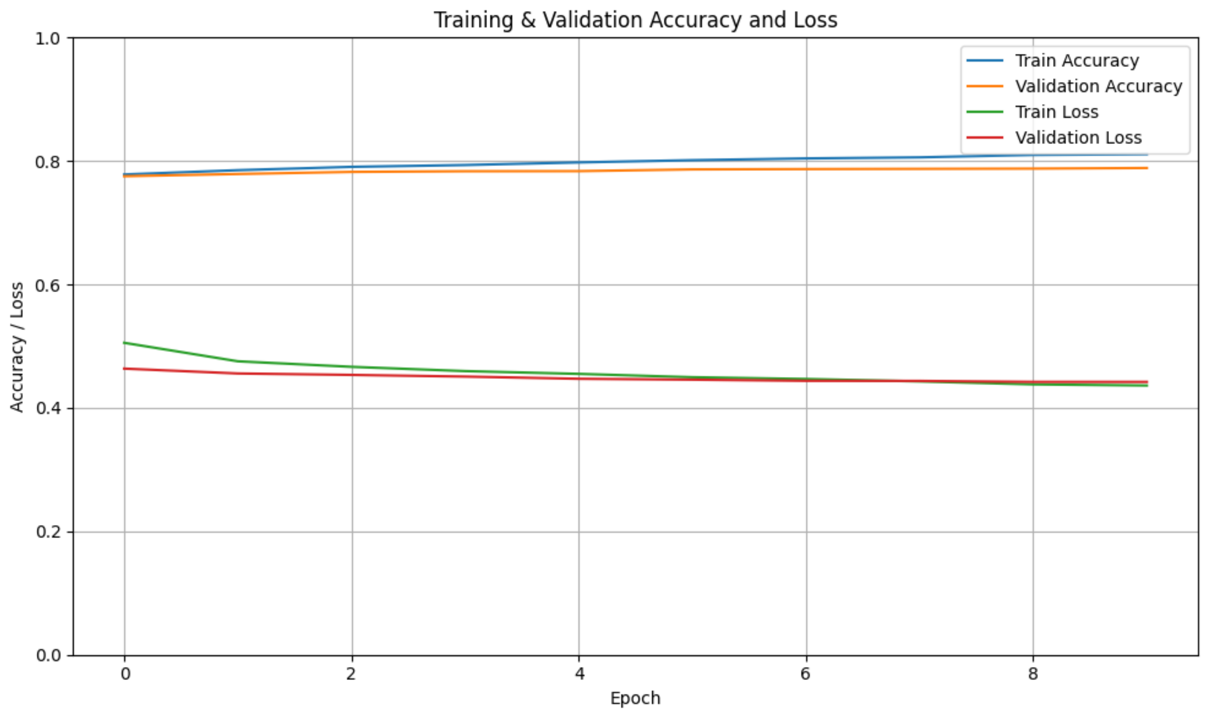

Figure 10: Dropout and Batch Norm Model

6. Epoch Increase and Early Stopping:
   To ensure the model had sufficient opportunity to converge, the maximum number of training epochs was increased from the initial 10 to 30. Also, early stopping was implemented to prevent potential overfitting and reduce unnecessary computation. This monitors the validation accuracy and stops the training process if no improvement is observed for 6 consecutive epochs.

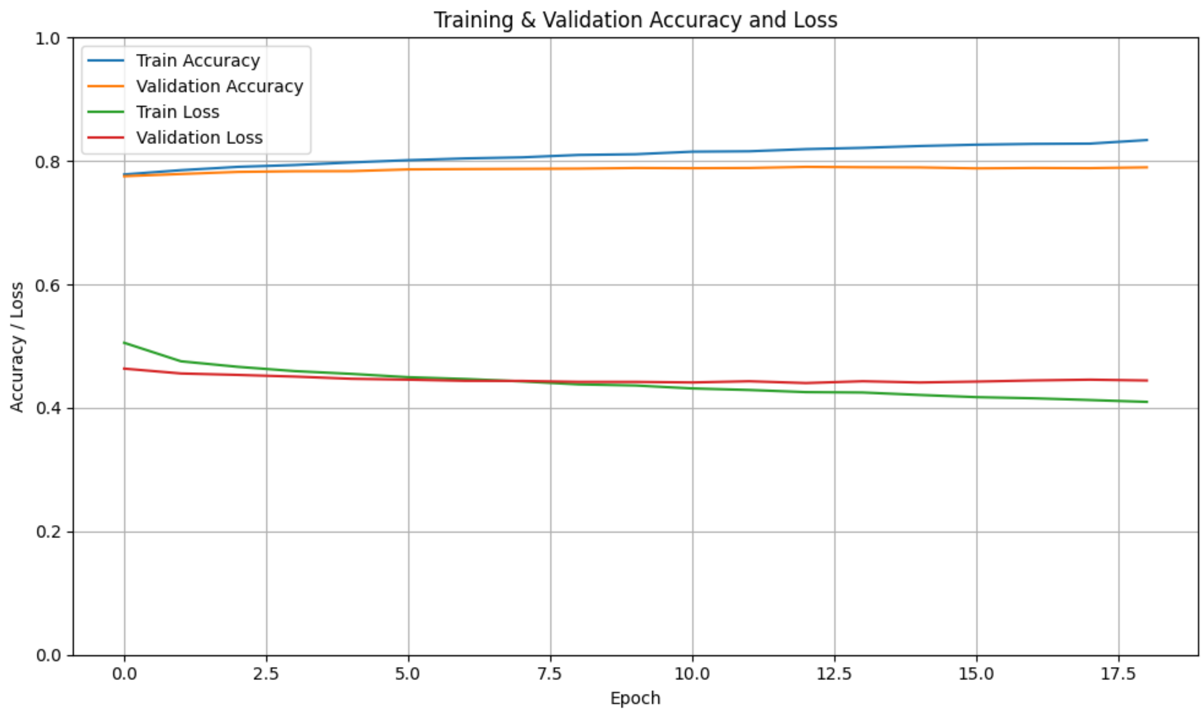

Figure 11: Increased Epoch and Early Stopping Model

7. Optimizer and Scheduler Experiments:
   Finally, the optimization algorithm was reviewed, comparing the standard Adam optimizer against AdamW, which modifies weight decay handling. In our experiments with the current model configuration, both optimizers yielded virtually identical performance metrics (Accuracy: 0.7915, F1: 0.7880). This similarity likely arose because the specific weight decay implementation difference between Adam and AdamW did not significantly impact this particular model setup, possibly due to the lack of explicit weight decay being applied or the model's nature. Given no performance advantage and Adam's common usage, the standard Adam optimizer was retained for the final tuning phase. Additionally, a ReduceLROnPlateau learning rate scheduler was employed to improve convergence stability. This scheduler monitored validation accuracy and reduced the learning rate by a factor of 0.1 if no improvement of at least 0.5% was observed over three consecutive epochs, helping to fine-tune the learning dynamics as training plateaued.

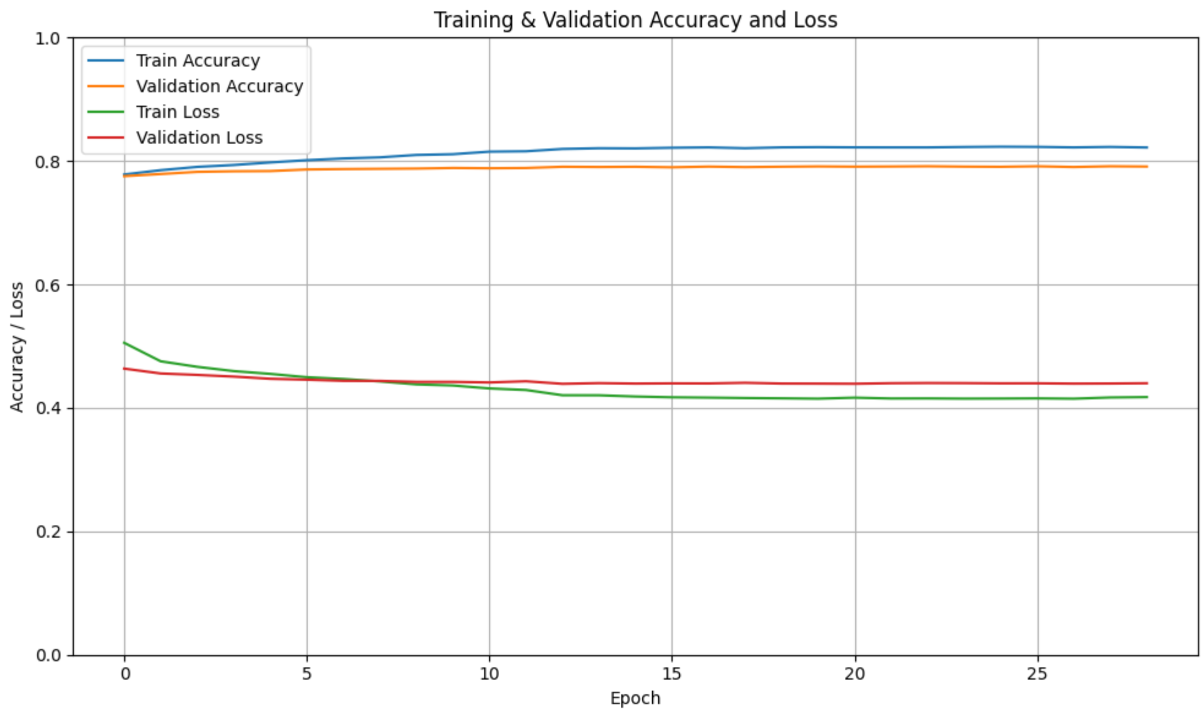

Figure 12: Adam Optimizer and Scheduler Model

***3.1.1. Table of trials.*** The table of trials shows a summary of the results from the experiments thoroughly analysed above. Each row highlights the main change we tested at that specific step. The accuracy scores listed are for the complete model setup at that point, including the new change.

| Trial | Description | Train Accuracy | Validation Accuracy |
|---|---|---|---|
| 1 | Baseline Model | 75.40 | 75.23 |
| 2 | glove-twitter-200 Embedding | 77.35 | 77.36 |
| 3 | 2 Layer Architecture | 78.95 | 78.20 |
| 4 | 3 Layer Architecture | 79.38 | 78.46 |
| 5 | 4 Layer Architecture | 80.28 | 78.29 |
| 7 | ReLU Activation Function | 79.38 | 78.46 |
| 7 | GELU Activation Function | 77.33 | 77.30 |
| 8 | LeakyReLU Activation Function | 79.38 | 78.34 |
| 9 | Tanh Activation Function | 80.14 | 78.48 |
| 10 | 0.2, 0.25, and 0.3 Dropout | ≈77.2 | ≈77.2 |
| 11 | 0.3 Dropout and Batch Normalization | 81.10 | 78.87 |
| 12 | Epoch Increase and Early Stopping | 81.57 | 79.05 |
| 13 | Adam Optimizer and Scheduler | 82.20 | 79.15 |
| 14 | AdamW Optimizer and Scheduler | 82.18 | 79.15 |
| 14 | Switch to ReLU | 80.10 | 78.48 |
| 15 | Switch to Tanh | 78.88 | 77.93 |
| 16 | Batch Size Increase to 128 | 81.21 | 79.11 |
| 17 | Optuna Optimization and Final Changes | 82.64 | 79.20 |

Table 1: Trials

### 3.2. Hyper-parameter tuning

The model configuration and training process involved the following key components and hyper-parameters:

- glove-twitter-200 Pretrained Embeddings:
  Initialized the embedding layer using glove-twitter-200 pretrained embeddings, specifically selected because their training on Twitter data aligns well with the context of the assignment's dataset.

- Three Hidden Layers (500, 201, 65 neurons):
  Optuna hyperparameter optimization trials determined the use of three hidden layers with neuron counts of 500, 201, and 65 respectively.

- GELU Activation Function:
  Applied the GELU activation function after each hidden layer for non-linearity. Although initial tests showed decreased perfomance, GELU significantly outperformed other activation functions once combined with the model's other features and regularization techniques.

- 0.3 Dropout and Batch Normalization:
  experiments showed dropout alone performed poorly, but the combination effectively regularized the model against overfitting and improved training stability.

- 128 Batch Size:
  One of the last parameters tuned, setting the batch size to 128 yielded a slight improvement, likely by providing a more stable gradient estimate for weight updates without significantly impacting computational efficiency.

- BCEWithLogitsLoss Function:
  Chose BCEWithLogitsLoss as the loss function, which is standard for binary classification tasks and combines a Sigmoid activation layer with Binary Cross Entropy loss in a numerically stable way.

- Adam Optimizer:
  Utilized the Adam optimizer; tests comparing it against AdamW showed nearly identical performance on the validation set, so the standard Adam was retained for this configuration.

- ReduceLROnPlateau Scheduler:
  Implemented the ReduceLROnPlateau learning rate scheduler to automatically decrease the learning rate when the validation loss plateaued, allowing the model to fine-tune weights more effectively during later stages of training.

- 30 Epochs with Early Stopping:
  Set a maximum training duration of 30 epochs while incorporating an early stopping mechanism monitoring validation loss. This prevents overfitting by stopping training when performance on unseen data no longer improves, saving computational time.

**Overfitting and Undefitting**

Overfitting means creating a model that matches (memorizes) the training set so closely that the model fails to make correct predictions on new data. On a plot like this, overfitting is typically indicated by the training accuracy continuing to rise while the validation accuracy flattens or drops, and/or the training loss continuing to decrease while the validation loss starts to increase. Looking at the graph, (see Figure 14) the gap between training and validation accuracy/loss remains relatively small and stable after the initial epochs, and the validation loss does not significantly increase. This suggests that the model is not suffering from excessive overfitting; it generalizes reasonably well to the validation data.

Underfitting occurs when a model is too simple to capture the underlying patterns in the data, resulting in poor performance on both the training and validation sets. This would be shown on the plot by both training and validation accuracy remaining low, or both training and validation loss remaining high, with neither showing much improvement over epochs. Looking at the graph, (see Figure 14) the model clearly learns, as both training and validation accuracy increase significantly from the start (reaching around 83% and 79% respectively) and both losses decrease. Therefore, the model does not appear to be underfitting the data.

### 3.3. Optimization techniques

Several optimization techniques were employed to tune the model and improve training efficiency and overall performance:

- Manual Tuning and Component Selection:
  Initial choices for components like the GELU activation, Adam optimizer, BCEWithLogitsLoss and architectural elements (e.g., depth and width of neural network, use of batch normalization) were based on common practices that tend to improve deep learning models. Parameters like the maximum number of epochs,

batch size, and scheduler settings (patience, factor) were also initially set based on typical values and adjusted manually.

- Automated Hyperparameter Optimization (Optuna):
  The Optuna framework was utilized for systematic hyperparameter search. Multiple studies were conducted to explore optimal values for key parameters, including the number of neurons in hidden layers, dropout probability, and learning rate, guiding the final selection of these values by efficiently searching the parameter space. The results from Optuna optimization are commented out at the bottom of my project's code submission.

- Learning Rate Scheduling:
  A ReduceLROnPlateau scheduler was implemented to adaptively adjust the learning rate during training, reducing it when validation accuracy plateaued, which helps in converging more precisely towards an optimal solution.

- Early Stopping:
  An early stopping mechanism monitored validation accuracy, stopping the training process if no improvement was observed for a set number of epochs. This prevented overfitting and saved computational resources by stopping unnecessary training iterations.

- Regularization Combination (Dropout and Batch Normalization):
  While tuned via Optuna and manual adjustments, the combined use of Dropout and Batch Normalization served as an optimization strategy to improve model generalization and stabilize the training process.

- (Attempted) GPU Acceleration:
  Attempts were made to leverage GPU acceleration using PyTorch's CUDA capabilities to speed up the computationally intensive training and Optuna search processes, although the final reported experiments were conducted on a CPU environment.

### 3.4. Evaluation

To assess the performance of the trained logistic regression model, I used several classification metrics, including accuracy, precision, recall, and F1-score. These metrics provide insight into the model's effectiveness in predicting sentiment labels.

#### Evaluation Metrics

- Accuracy: Measures the proportion of correctly classified instances out of the total instances.
  $$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$
  The validation accuracy achieved is 0.7920, meaning the model correctly predicts around 79.20% of the samples.

- Precision: Measures the accuracy of positive predictions
  $$Precision = \frac{TP}{TP + FP}$$

The precision is 0.7993, indicating that when the model predicts a positive sentiment, it is correct 79.93% of the time.

- Recall: Measures the ability to capture all relevant instances

$$Recall = \frac{TP}{TP + FN}$$

The recall achieved is 0.7797, meaning the model correctly identifies 77.97% of actual positive samples.

- F1 Score: Balances precision and recall into a single metric

$$F1Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The F1-score is 0.7894, reflecting a balanced trade-off between precision and recall.

**Validation Set Performance**

The detailed classification report for both classes is as follows:

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| 0 (Negative) | 0.78 | 0.80 | 0.79 | 21197 |
| 1 (Positive) | 0.80 | 0.78 | 0.79 | 21199 |
| accuracy | | | 0.79 | 42396 |
| macro avg | 0.79 | 0.79 | 0.79 | 42396 |
| weighted avg | 0.79 | 0.79 | 0.79 | 42396 |

Table 2: Classification Report

***3.4.1. ROC curve.*** The Receiver Operating Characteristic curve, or ROC curve illustrates the performance of a binary classifier model at varying threshold values

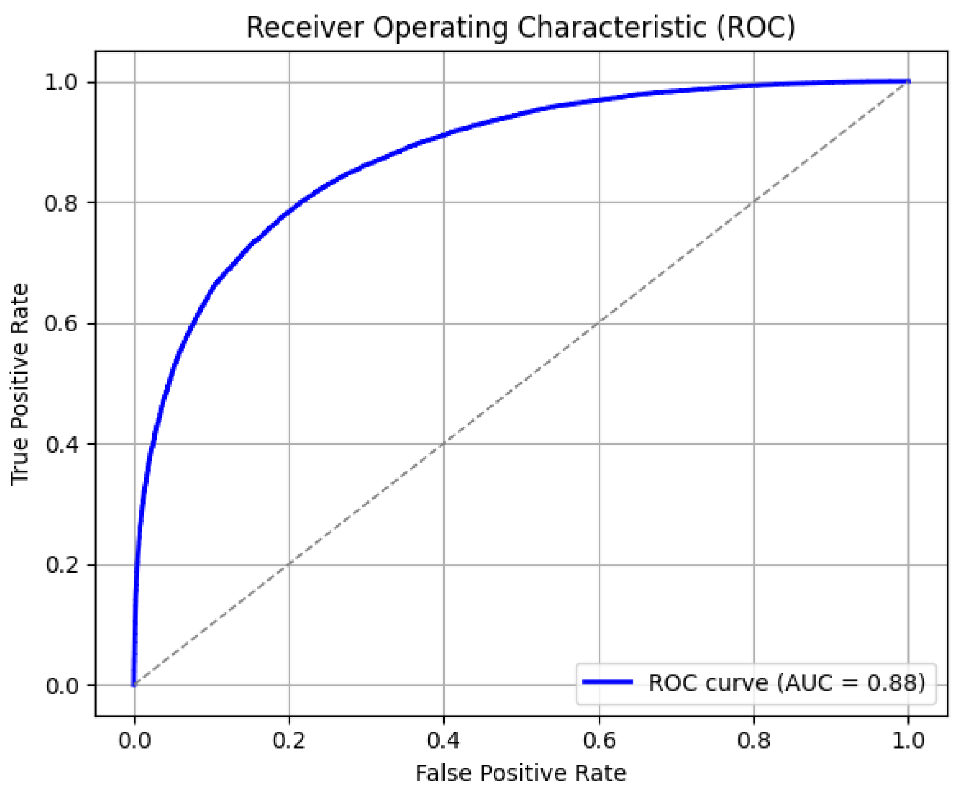

Figure 13: ROC curve

***3.4.2. Learning Curve.*** The Learning Curve for Logistic Regression which was analysed in a previous section.

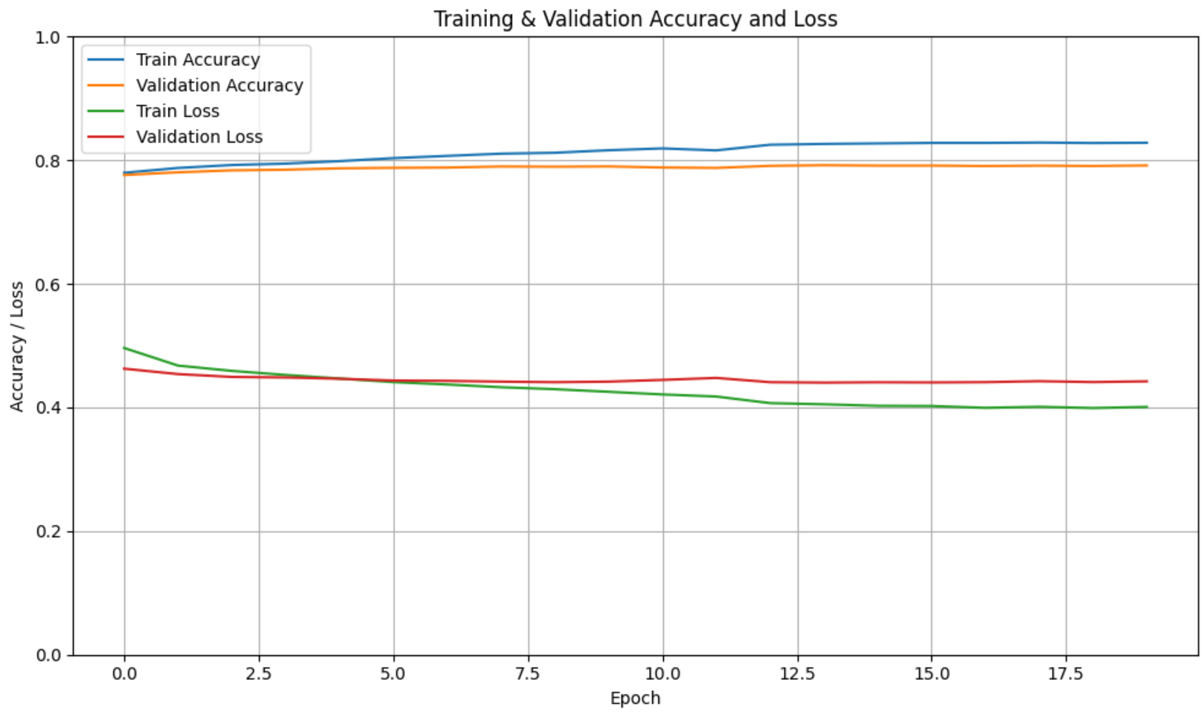

Figure 14: Learning Curve

***3.4.3. Confusion matrix.*** The confusion matrix is a performance evaluation tool in machine learning, representing the accuracy of a classification model. It displays the number of true positives, true negatives, false positives, and false negatives.

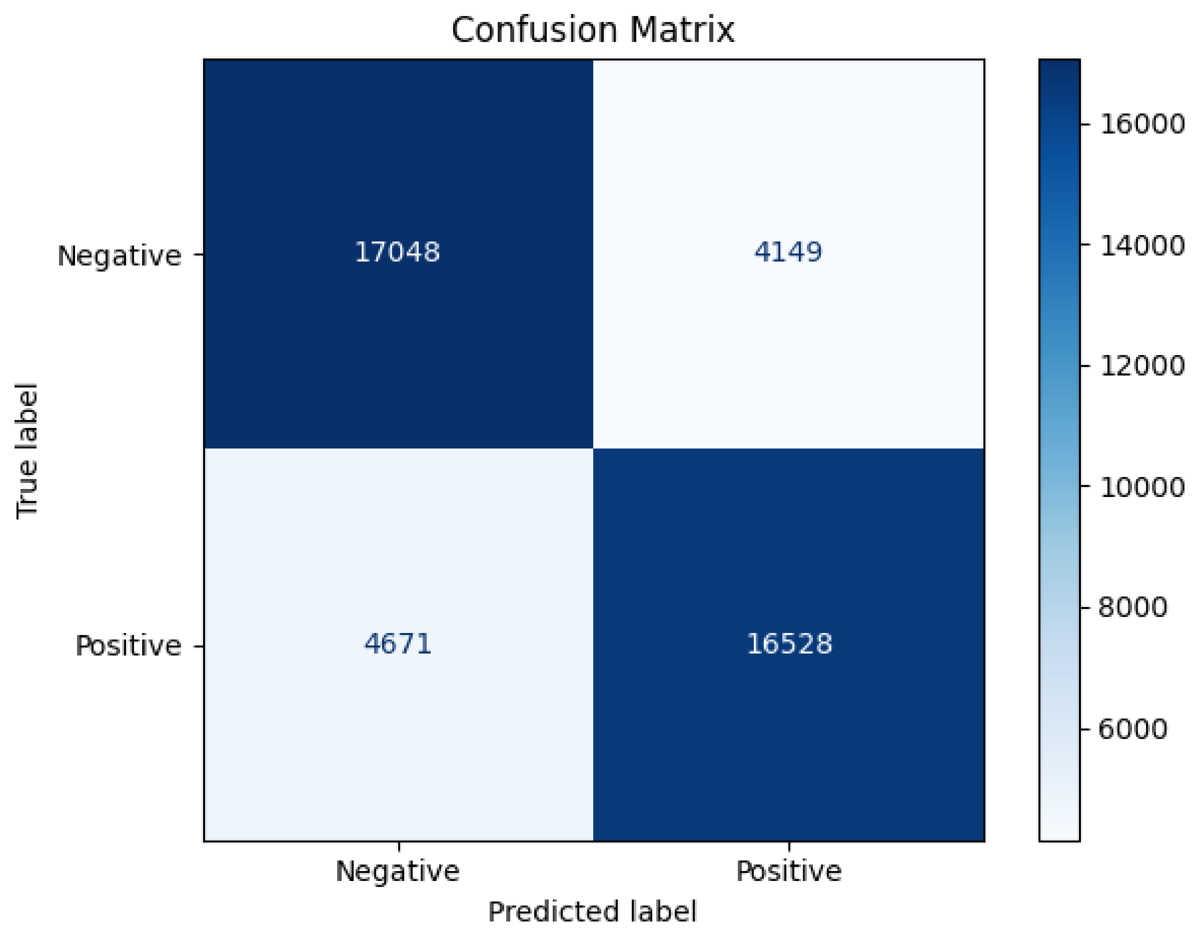

Figure 15: Confusion matrix

## 4. Results and Overall Analysis

### 4.1. Results Analysis

The objective of this work was to develop a deep learning model for sentiment classification on tweet data. The final model configuration, through iterative tuning and optimization, achieved the following performance metrics:

- Training Set Accuracy: 82.64%
- Validation Set Accuracy: 79.20%
- Test Set Accuracy (Kaggle): 78.98%

These results indicate that the model learned effectively from the training data and generalized reasonably well to unseen data. The gap between training accuracy (82.64%) and validation/test accuracy (79.20%) is relatively small, suggesting that the regularization techniques and early stopping mechanism were successful in mitigating significant overfitting. Achieving a test accuracy of 78.98% demonstrates a functional classifier. This level of performance is considered acceptable and aligns closely with the results observed from many other submissions within the context of this specific assignment. The performance was likely within the expected range for a deep neural network relying on averaged static word embeddings like GloVe, which, while effective, may not capture the full complexity of language compared to more advanced architectures.

There are several additional experiments and improvements that could be explored to further enhance the model's performance such as:

- Further Optimization:
  Although Optuna was used, expanding the search space or ranges would allow further testing of the hyperparameters possibly resulting in increased perfomance.

- Out-of-Vocabulary (OOV) Word Handling:
  Analyze the impact of OOV words and experiment with more advanced transformer based strategies, training custom embeddings on the task-specific corpus, or initializing OOV words with random vectors to see how performance is affected.

- Advanced Architectures:
  Explore models inherently designed for sequential data. Recurrent Neural Networks (RNNs) could capture temporal dependencies better than the current feedforward approach. Furthermore, investigating Transformer-based models would likely yield significant performance improvements, as they excel at capturing contextual information.

#### *4.1.1. Best trial.*

- Performance Metrics
  - Training Accuracy: 82.64%
  - Test Accuracy (Kaggle): 78.98%
  - Validation Accuracy: 79.20%
  - Validation Precision: 79.93%
  - Validation Recall: 77.97%
  - Validation F1-Score: 78.94%
- Model
  - sentiment classifier using deep neural networks for the English-language Twitter dataset, using PyTorch and Word2Vec word embeddings
- Best Hyperparameters
  - glove-twitter-200 Pretrained Embeddings
  - Three Hidden Layers (500, 201, 65 neurons)

- GELU Activation Function
- 0.3 Dropout and Batch Normalization
- 128 Batch Size
- BCEWithLogitsLoss Function
- Adam Optimizer
- ReduceLROnPlateau Scheduler
- 30 Epochs with Early Stopping

### 4.2. Comparison with the first project

This project explored a deep learning approach for sentiment classification, contrasting with the methods used in the first project which relied on a traditional machine learning pipeline including Logistic Regression and TF-IDF.

Interestingly, the simpler Logistic Regression model with TF-IDF features slightly outperformed the more complex deep neural network using averaged GloVe embeddings on both the validation and test sets for this specific task and dataset. While the neural network has the theoretical capacity to learn more complex patterns, the TF-IDF approach, which directly leverages word frequencies and discriminative terms, proved more effective here. The loss of information caused by averaging GloVe embeddings might have hindered the neural network's ability to capture nuances essential for slightly better performance. The deep learning model, despite its complexity and extensive tuning, did not yield superior results in this instance, highlighting that more complex models are not always better and that feature representation is critical.

## 5. Bibliography

## References

[1] Dan Jurafsky and James H. Martin. *Speech and Language Processing*.

[2] Manolis Koubarakis. Backpropagation, 2025.

[3] Manolis Koubarakis. Feed forward neural networks, 2025.

[4] Manolis Koubarakis. Perceptrons, 2025.

[5] Manolis Koubarakis. Training dnns, 2025.

[6] Manolis Koubarakis. Word vectors, 2025.

[3] [4] [2] [5] [6] [1]