UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP

Student name: *Maria-Malamati Papadopoulou*
sdi: sdi2200134

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

Contents

1. Abstract

The objective of this assignment is to build a sentiment classification model using a Deep Neural Network, trained on a Twitter dataset with binary sentiment labels, positive (1) and negative (0). To approach this task, I begin by thoroughly preprocessing the tweets to normalize informal language and clean up noise typical of social media content. Then, I use pre-trained GloVe embeddings to convert each tweet into a fixed-size numerical representation that captures semantic information. These embeddings are used as input to a multi-layer neural network built in PyTorch. The model's performance is evaluated using accuracy, precision, recall, and F1-score on a validation set and test set. Throughout the process, I use learning curves and performance metrics to monitor training, apply early stopping to prevent overfitting, and perform hyperparameter tuning to optimize results.

2. Data processing and analysis

2.1. Pre-processing

Before utilizing GloVe embeddings and training the deep neural network for sentiment analysis, careful pre-processing of the text data is crucial. Proper pre-processing enhances the quality of word representations and significantly improves the model's ability to learn meaningful patterns. The main pre-processing steps I applied are the following:

1. Lowercasing: Converting all text to lowercase ensures that words like "The" and "the" are treated the same, preventing unnecessary duplication in the vocabulary and promoting more consistent word embeddings.

2. Tokenization: I performed tokenization tailored to the structure of Twitter text. This involves splitting sentences into words and symbols, considering special Twitter elements like mentions, hashtags, and emojis. Precise tokenization is essential for accurately capturing the semantic units of tweets.

3. Contractions: Contractions, like "isn't" to "is not" were expanded to their full forms. This increases clarity in the data and ensures that pre-trained GloVe embeddings can correctly map words without missing representations for shortened forms.

4. Removing mentions, hashtags and links: Eliminating Twitter mentions (@user) for complete anonymization, hashtags and URLs to focus on the actual tweet content due to analysis' results.

5. Removing Special Characters and Numbers: Filtering out punctuation, non-ASCII characters, and numbers to reduce noise in the data.

6. Normalizing repeated characters and spaces: Twitter language often includes elongated words for emphasis like "plzzzz" instead of "please". I normalized these cases by reducing repeated characters to a single or standard form, for example, "plzzzz" to "plz". Similarly, redundant spaces were standardized to a single space to preserve the structural uniformity of the data.

7. Slang normalization: Common internet slang and abbreviations were mapped to their standard English equivalents. This improves the semantic interpretability of the tweets and ensures that the pre-trained embeddings can correctly process the normalized words.

Each of these pre-processing steps contributes to reducing the variability and noise inherent in user-generated content like tweets. By normalizing the text, we create a cleaner and more consistent dataset, allowing the neural network to focus on the true sentiment patterns rather than memorizing idiosyncrasies of informal language.

2.2. Analysis

To gain a comprehensive understanding of the dataset, I first computed the number of unique words across the training data, revealing approximately 234,492 distinct features (see Figure 1). This high vocabulary size highlights the linguistic diversity typical of user-generated content like tweets, which often include slang, typos, informal contractions, and creative spelling variations.
Next, I examined the class distribution within the training set. Out of a total of 148,388 tweets, the distribution was approximately even, with positive and negative tweets each comprising around 50% of the data (see Figure 2). As a result, the training dataset exhibits a balanced class distribution and this ensures that the model does not suffer from class imbalance, which could have led to biased predictions favoring the majority class.
Similarly, analysis of the validation set, consisting of 42,396 tweets, also indicated a balanced 50/50 split between positive and negative sentiments (see Figure 3). The balanced nature of both the training and validation datasets ensures that the model's performance metrics will be more reliable and not artificially inflated due to a dominant class.
These insights from the previous project helped refine the preprocessing steps and provided an initial understanding of how sentiment is expressed in the dataset. EDA was conducted exclusively on the training dataset to prevent any potential data leakage from the validation and test dataset, thereby ensuring the integrity of the evaluation process and maintaining unbiased performance assessment. Additionally, since the model is trained solely on the training dataset, focusing the analysis on this subset ensures that the extracted insights and preprocessing choices are directly relevant to what the model will learn. This makes the EDA more representative and meaningful for guiding model development.

2.3. Data partitioning for train, test and validation

To effectively train and evaluate the sentiment classification model, the dataset was partitioned into three subsets: training set, validation set, and test set, as it was shown in the lectures. The dataset was split as follows:

1. Training Set (X_train, y_train): This set contains 148,388 tweets, as identified during the Exploratory Data Analysis (EDA). It is used to fit the model, allowing it to learn patterns, word associations, and sentiment relationships from labeled

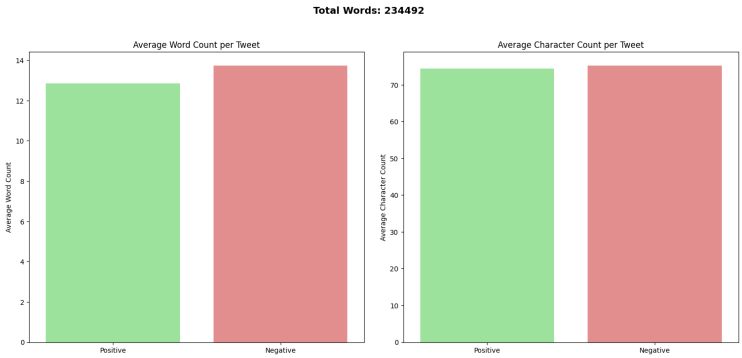

Figure 1: Average Word and Character Count per Tweet

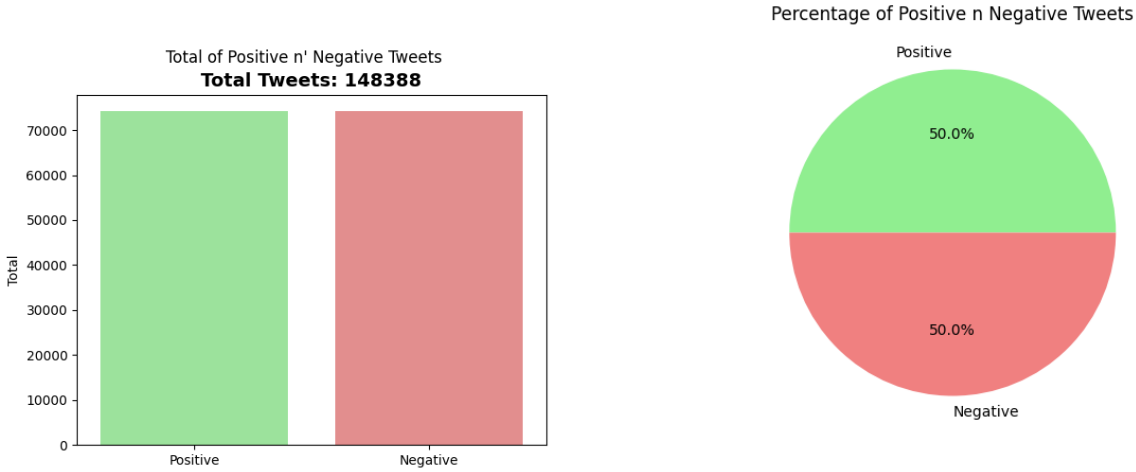

Figure 2: Comparison of Positive and Negative Tweets in Training Set

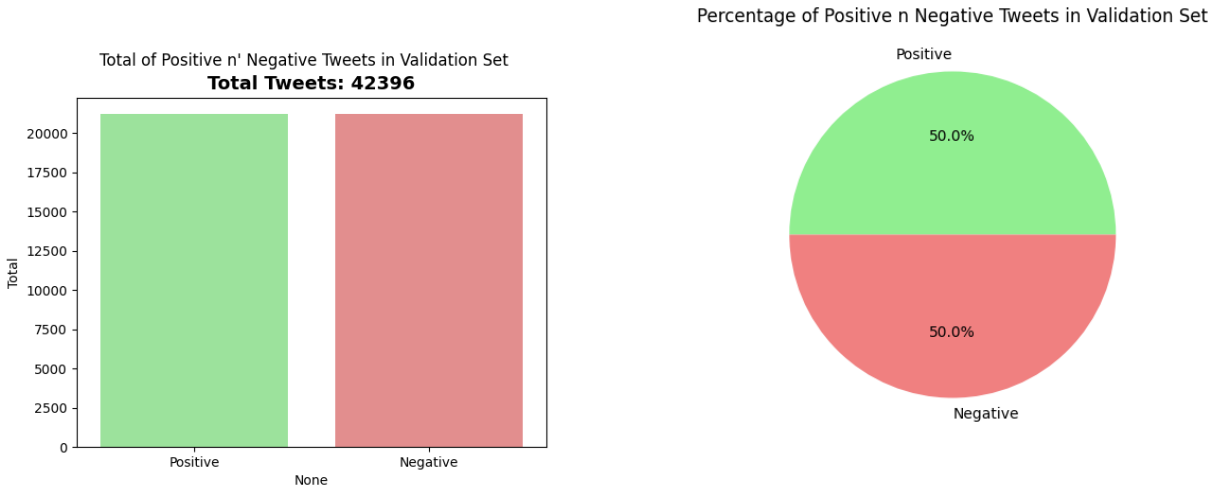

Figure 3: Comparison of Positive and Negative Tweets in Validation Set

examples. A sufficiently large training set ensures that the model can generalize well without underfitting.

2. Validation Set (X_val, y_val): Comprising 42,396 tweets, the validation set is employed to fine-tune the model's hyperparameters, monitor performance during training, and prevent overfitting. Validation performance offers an early estimate of the model's generalization ability, guiding choices such as learning rate adjustments, model architecture refinements, or early stopping/ dropout decisions.

3. Test Set (X_test): The test set consists of a separate, unseen portion of the data that was not utilized during model training or validation. It serves as the final benchmark to evaluate the model's real-world performance and assess its ability to generalize to completely new inputs.

2.4. Vectorization

To transform the raw textual data into a numerical format suitable for deep neural network training, I employed pre-trained GloVe embeddings, specifically the **glove.twitter.27B.200d model**. These embeddings have been trained on a massive corpus of Twitter data, making them particularly well-suited for capturing both semantic meanings and the informal sentiment expressions frequently encountered in tweets (such as slang, abbreviations, and emoticons). These embeddings were then converted into Word2Vec format to enable efficient loading and compatibility with the Gensim library for feature extraction.
Each tweet is first tokenized using the TweetTokenizer from the NLTK library. This tokenizer is optimized for handling the idiosyncrasies of social media text, including mentions (@user), hashtags (#topic), emojis, and informal language patterns. Then, for each token (word) in a tweet, its corresponding GloVe vector is retrieved, provided that the word exists in the GloVe vocabulary. Words who not present in the embeddings are ignored, ensuring that only valid semantic information contributes to the representation. The final vector representation of each tweet is computed by taking the average of all available word embeddings within the tweet. This produces a fixed-size 200-dimensional vector for every tweet, regardless of its original length.

This technique of averaging word embeddings offers several advantages:

1. It provides a compact and dense representation of the text, keeping the input size manageable for the neural network.

2. Despite its simplicity, averaging retains key semantic information, allowing the model to capture the overall meaning and sentiment of the tweet effectively.

This simple yet effective technique of averaging word vectors enables the model to process semantic information while keeping the input representation compact and computationally efficient.

3. Algorithms and Experiments

3.1. Experiments

1. Baseline Experiment: Simple Deep Neural Network
   As a first step, I implemented a baseline Deep Neural Network to establish a reference point for future improvements. The architecture was deliberately kept simple, consisting of a single hidden layer with a linear activation function. Additionally, no regularization techniques such as Dropout or Batch Normalization were applied.
   For training, I used the setup provided in the course tutorial: a learning rate of 1e-4, Mean Squared Error (MSE) as the loss function, and the Stochastic Gradient Descent (SGD) optimizer. The model was trained with a batch size of 32 for 20 epochs, without any form of early stopping.
   This initial configuration served two purposes: to familiarize myself with the training process of DNNs in PyTorch and to provide a performance baseline. The model achieved a validation accuracy of approximately 52%, which is only marginally better than random guessing for a binary classification task. So, this baseline experiment emphasized the importance of proper model architecture, activation functions, and regularization techniques and it motivated a shift toward a more expressive network with better-suited training strategies in subsequent experiments.

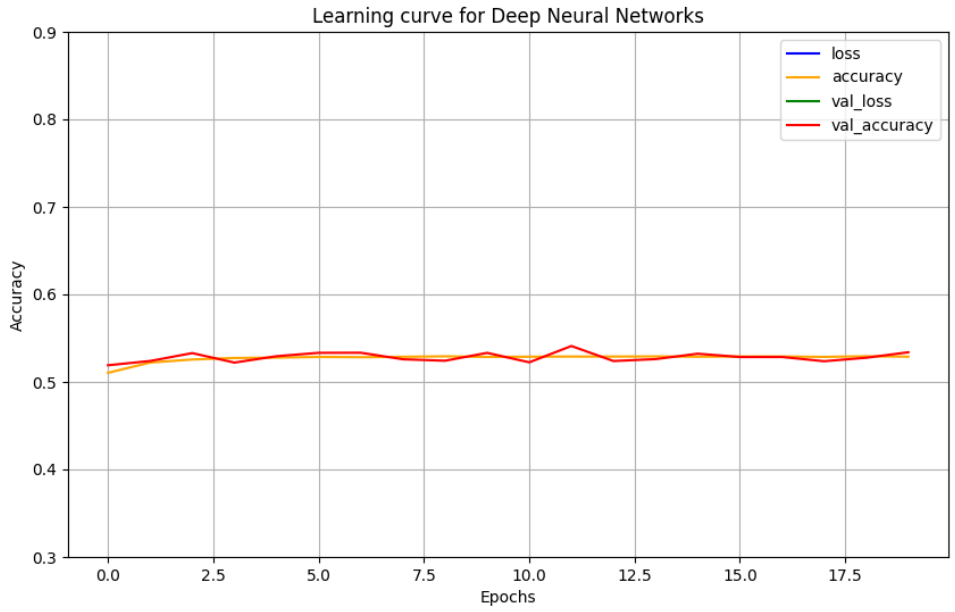

Figure 4: Baseline Model

2. Increasing Epochs and Applying Early Stopping
   For the first optimization attempt, I extended the number of training epochs from 20 to 50, allowing the model more opportunities to learn from the data and potentially improve its ability to generalize. However, training for too many epochs can lead to overfitting, where the model memorizes the training data rather than learning generalizable patterns.
   To mitigate this risk, I introduced early stopping to the training loop. This tech-

nique monitors the validation loss and halts training when no further improvement is observed for a predefined number of epochs (patience). It provides a balance between underfitting (too few epochs) and overfitting (too many epochs), ensuring that training stops at the optimal point of model performance on unseen data.
This adjustment led to an improvement in validation accuracy, increasing it to 53.46%. The results demonstrate that allowing more training iterations, when controlled by early stopping, can significantly enhance the model's ability to generalize. It also confirmed that some of the shortcomings in the baseline model were due to premature termination of training.

3. Tuning the Model's Architecture – Activation Functions
Following improvements in training strategy, I focused on experimenting with the architecture of the Deep Neural Network, specifically the activation function, which plays a critical role in enabling the network to learn non-linear relationships in the data.
Initially, the model used a linear activation function, which limits expressiveness and prevents the network from modeling complex patterns. To address this, I replaced the activation with ReLU, a widely-used function known for mitigating the vanishing gradients problem and promoting sparse activations. This change led to a noticeable improvement in validation accuracy to 56.26%, indicating better learning and generalization.
In pursuit of further improvements, I also experimented with SELU and Leaky ReLU. SELU is designed to encourage self-normalizing networks, while Leaky ReLU aims to fix the issue of "dying ReLUs" by allowing a small gradient for negative inputs. However, in practice, neither of these alternatives improved model performance compared to standard ReLU. This may be due to the relatively simple structure of the network or the nature of the dataset, where ReLU's behavior proved sufficient. As a result, ReLU's simplicity and effectiveness made it the best fit in this context.

4. Loss Function and Batch Size Tuning
Continuing with training-related hyperparameters, I investigated the impact of the loss function and batch size on model performance.
In the initial baseline, the model used Mean Squared Error (MSE) as the loss function, mainly because it was present in the tutorial code. However, MSE is not ideal for classification tasks, especially binary classification, as it assumes continuous targets and tends to produce less informative gradients for probability-based predictions. To address this, I switched to loss functions better suited for binary classification. I first tried Binary Cross Entropy (BCE) and then BCEWithLogitsLoss(), which is a more numerically stable version that combines a sigmoid layer and the binary cross-entropy loss into one function. This is particularly helpful when dealing with logits directly as model outputs, avoiding precision issues when applying sigmoid separately. Among the two, BCEWithLogitsLoss provided superior validation accuracy of 67.18%, and became the standard in subsequent experiments.
I also tuned the batch size, starting from the baseline of 32. Increasing it to 64, and then to 128, revealed that a batch size of 128 provided the best performance in terms of both training stability and generalization. Larger batch sizes tend

to smooth out noisy gradient estimates and leverage parallel hardware more efficiently, though they can sometimes hinder generalization if too large. In this case, 128 was a good balance and succeed 68.43% accuracy in validation data.

5. Optimizer and Learning Rate Tuning
   In this stage, I focused on optimizing two of the most critical hyperparameters that govern the training dynamics of deep neural networks: the optimizer and the learning rate.
   Initially, I used Stochastic Gradient Descent (SGD), a traditional optimizer known for its simplicity but also for its slow convergence rate, especially in high-dimensional optimization problems. This was likely a contributing factor to the modest performance in early experiments. To improve convergence speed and optimization stability, I experimented with two adaptive optimizers: Adam and AdamW. Adam combines the benefits of RMSProp and momentum, automatically adjusting learning rates for each parameter based on first and second moment estimates. AdamW is a variation that decouples weight decay from the gradient update, offering improved generalization under certain regularization regimes. I set the weight decay parameter in AdamW to a small value 1e-2, which is commonly recommended in literature for modest regularization. Empirically, Adam outperformed both SGD and AdamW in my experiments, yielding faster convergence and better validation accuracy. This suggests that for this specific task, where the dataset is relatively small and noisy, Adam's adaptive updates provide a significant advantage and improves accuracy of validation dataset in 77.89%.
   In parallel, I also tuned the learning rate, a crucial hyperparameter that directly affects the convergence speed and stability of training. Using a learning rate of 1e-3 resulted in unstable training and overshooting and worse accuracy of 77.15%, while 1e-5 was too small to make meaningful progress within a reasonable number of epochs. A learning rate of 1e-4 provided the best balance, allowing the model to converge steadily while maintaining generalization.
   These conclusions were further validated through hyperparameter optimization with Optuna, which also identified Adam and a learning rate close to 1e-4 as optimal choices, which resulted in accuracy of 77.89% in the validation dataset.

6. Varying the Number of Hidden Layers
   To explore how network depth affects model performance, I experimented with increasing the number of hidden layers. Starting from the baseline architecture with a single hidden layer, I tested networks with 2, 3, and 4 hidden layers to assess the trade-off between model capacity and overfitting.
   The results showed that the best validation accuracy, approximately 78.61%, was achieved using 3 hidden layers. In comparison, the model with 2 hidden layers reached only about 77.55% accuracy, and increasing to 4 layers slightly reduced performance to around 78.05%.
   These findings suggest that depth helps the model capture more complex patterns, but there is a point beyond which adding more layers offers diminishing returns or even degrades performance. In this case, 4 layers may have introduced unnecessary complexity, leading to overfitting or unstable training, especially given the relatively small and noisy nature of Twitter data.

7. Applying Dropout and Batch Normalization for Regularization

To further improve model performance and address overfitting, I incorporated two well-established regularization techniques: Dropout and Batch Normalization, both of which were discussed in the course lectures.
First, I introduced Dropout with a rate of 0.3 between layers. Dropout randomly deactivates a fraction of neurons during training, preventing the model from becoming overly reliant on specific paths through the network. Despite its simplicity, this technique often leads to better generalization by forcing the model to learn redundant representations. In my experiments, adding Dropout led to an increase in validation accuracy in 78.77%, which proved to be a significant gain and a clear indication of improved robustness.
In addition, I added Batch Normalization layers before each activation function. Batch Normalization normalizes the input of each layer across the mini-batch to have zero mean and unit variance, which helps stabilize and accelerate training. It also provides a mild regularizing effect by introducing noise through mini-batch statistics. This change not only improved convergence speed but also yielded a slight boost in validation performance, specifically reaching the highest accuracy of 79.09%.

8. Pretrained Word Embeddings
   In the final set of experiments, I focused on the pretrained word embeddings used for tweet vectorization, an essential component in capturing semantic meaning from text data.
   I began with GloVe 6B 50-dimensional embeddings (glove.6B.50d.txt), a general-purpose embedding trained on Wikipedia corpus. However, the relatively low dimensionality limited the model's ability to capture nuanced semantic and sentiment-related patterns in tweets.
   To address this, I switched to GloVe 6B 200-dimensional embeddings (glove.6B.200d.txt), which provided richer word representations. The increase in dimensionality allowed the model to better encode the contextual and sentiment-specific properties of the text, resulting in improved accuracy.
   Finally, I tested GloVe Twitter embeddings (glove.twitter.27B.200d.txt), which are trained on 27 billion tweets and specifically designed for social media content. This embedding set not only preserved the 200-dimensional richness but also aligned more closely with the linguistic characteristics of the dataset—such as slang, abbreviations, hashtags, and emojis. As a result, this configuration yielded the highest validation accuracy, confirming the importance of domain-specific embeddings in sentiment analysis.

3.1.1. Table of trials. The table below summarizes the key experiments conducted during model development, highlighting the impact of each architectural or training change on validation accuracy. These trials reflect a step-by-step approach to incrementally improve the model's performance through informed experimentation:

This progression shows how careful tuning of architecture, optimization strategy, and regularization significantly improved the model from a modest baseline of 52.38% to over 79.09% validation accuracy. The most impactful changes were the switch to Adam optimizer, using an appropriate loss function, and incorporating Dropout and Batch Normalization for regularization.

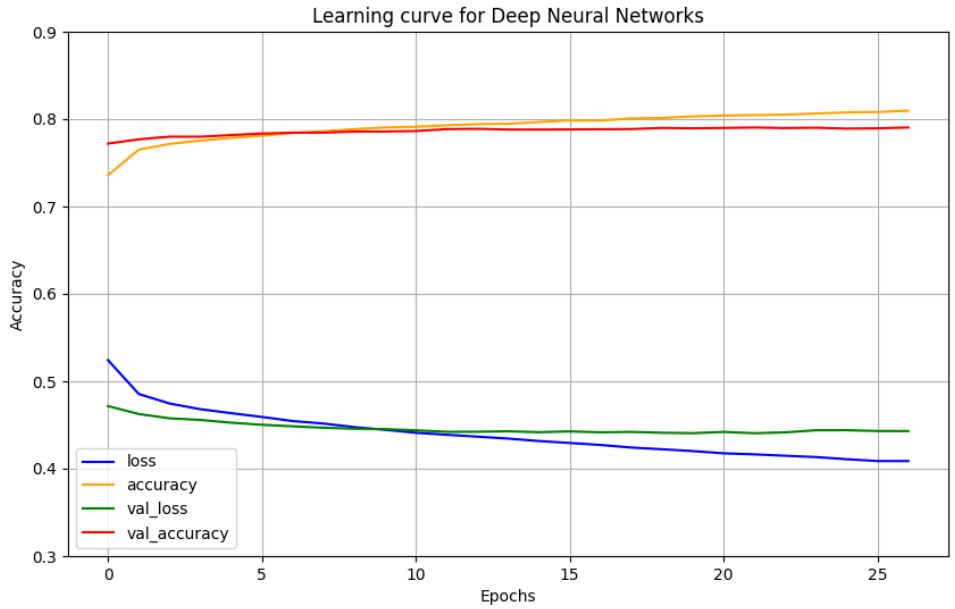

Figure 5: Final Model

Trial	Experiment	Score
0	Baseline Model (Tutorial code: 1 hidden layer, MSE loss, SGD optimizer)	52.38%
1	Increased training epochs to 50 and added early stopping	53.46%
2	Changed activation function to ReLU	56.26%
3	Switched loss function to Binary Cross Entropy (BCEWithLogitsLoss)	67.18%
4	Increased batch size to 128	68.43%
5	Replaced SGD with Adam optimizer (learning rate = 1e-4)	77.89%
6	Used 4 hidden layers (512, 256, 128, 32)	78.05%
7	Optimized architecture to 3 hidden layers (512, 256, 128)	78.61%
8	Applied Dropout (rate = 0.3)	78.77%
9	Added Batch Normalization before each activatio	79.09%

Table 1: Trials

3.2. Hyper-parameter tuning

The final model achieved an accuracy of 0.79096 on the validation dataset and 0.78994 on the test dataset, demonstrating strong and consistent performance across both seen and unseen data.
For this configuration, I used the glove.twitter.27B.200d.txt pretrained word embeddings, which were converted to Word2Vec format for compatibility with Gensim. These embeddings were ideal for this task due to their high dimensionality and training on Twitter-specific text, which closely matches the nature of the dataset. Prior to embedding, I applied an extensive text preprocessing function to normalize and clean the tweets. This step removed noise such as mentions, URLs, and informal language artifacts, ensuring the embeddings captured meaningful linguistic features.

After systematic experimentation and tuning, the optimal Deep Neural Network configuration was as follows:

- 3 Hidden Layers (512, 256, 128 neurons):
  This depth provided a strong balance between model expressiveness and generalization. Too few layers underfit the data, while too many led to overfitting. This architecture allowed the model to learn hierarchical representations of the input.

- Adam Optimizer:
  Among the tested optimizers, Adam offered the best performance. It adapts learning rates for each parameter based on estimates of first and second moments of the gradients, enabling fast and stable convergence.

- Learning Rate = 1e-4:
  This value proved to be the most suitable, large enough for efficient learning, yet small enough to avoid instability or overshooting during training.

- Loss Function = BCEWithLogitsLoss:
  This function is well-suited for binary classification, combining a sigmoid layer with binary cross-entropy loss in a numerically stable way.

- Batch Size = 128:
  After testing different values, a batch size of 128 provided the best trade-off between training stability and performance. It smoothed gradient estimates and enabled efficient GPU utilization.

- Dropout and Batch Normalization:
  Dropout with a rate of 0.3 helped reduce overfitting by randomly disabling neurons during training, forcing the model to generalize. Batch Normalization stabilized training and improved convergence by normalizing layer inputs.

- 50 Epochs with Early Stopping:
  The model was allowed to train for up to 50 epochs, but early stopping monitored validation loss and halted training when no further improvement was observed. This helped prevent overfitting and reduced training time.

Underfitting or Overfitting?

- Underfitting:
  Underfitting occurs when the model is too simple or poorly trained to capture the underlying patterns in the data. In the presented learning curve, there is no sign of underfitting because both training and validation accuracies are relatively high (around 79% - 80%), and the training loss steadily decreases. This indicates that the model has successfully learned meaningful features from the training data.

- Overfitting:
  Overfitting typically manifests as a widening gap between training and validation performance, where the model continues to improve on the training set while performance on the validation set stagnates or declines. In this case, a slight divergence appears after epoch 15, but it remains stable and controlled. The validation accuracy does not degrade significantly, suggesting that overfitting is minimal to zero. This stability is likely the result of applying Dropout, Batch Normalization, and Early Stopping, which collectively act as effective regularization techniques.

The learning curves suggest that the model is neither underfitting nor overfitting. Instead, it demonstrates good convergence, with a well-balanced trade-off between bias and variance. Early stopping appears to have halted training at an optimal point, just as performance on the validation set began to plateau. These results confirm that the chosen architecture, optimizer, and regularization methods are well-aligned with the complexity of the dataset and the sentiment classification task.

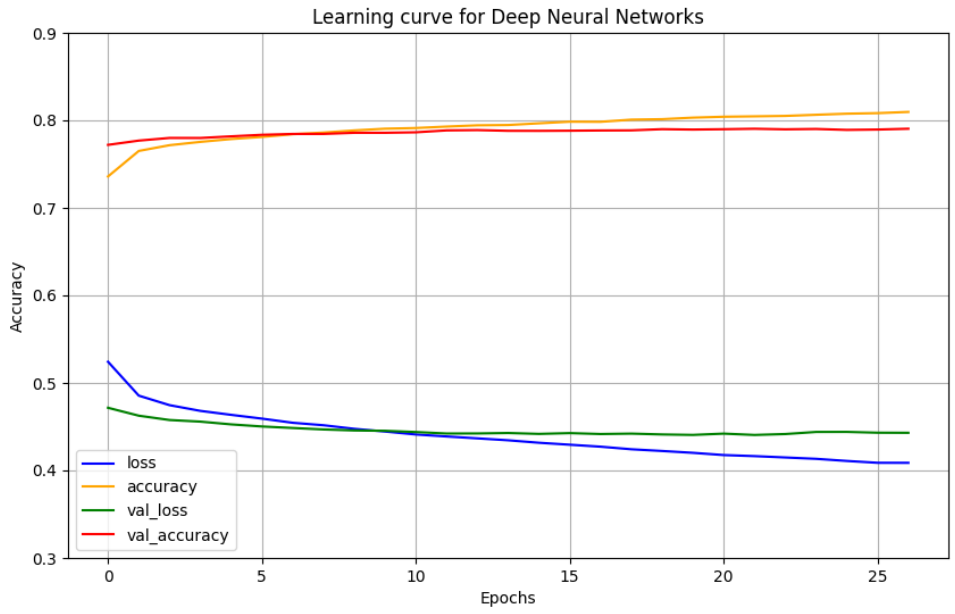

Figure 6: Final Model

3.3. Optimization techniques

To enhance the model's performance, a combination of manual experimentation and

automated hyperparameter optimization was employed throughout the development process.

Manual Experiments
Extensive manual tuning was carried out across several components of the model. Each of these contributed to incremental performance improvements:

- Text Preprocessing Function (primarily in Exercise 1):
  The preprocessing function was refined to handle noisy Twitter data effectively. This included steps such as lowercasing, tokenization, contraction expansion, removal of mentions, hashtags, URLs, and correction of common spelling errors. These improvements helped the embeddings better reflect the sentiment-carrying structure of each tweet.

- Pretrained Word Embeddings:
  Multiple GloVe embeddings were tested, glove.6B.50d, glove.6B.200d, and glove.twitter.27B.200d. The final choice was glove.twitter.27B.200d, which was specifically trained on Twitter data and provided the best performance due to its alignment with the domain and sufficient dimensionality.

- Architecture of the Deep Neural Network:
  I experimented with varying the number of hidden layers (1–4), neuron counts, and activation functions (ReLU, LeakyReLU, SELU). The optimal structure was a 3-layer architecture with sizes (512, 256, 128) and ReLU activations, which achieved the best balance of complexity and generalization.

- Training Hyperparameters:
  Systematic tuning was performed on the optimizer (SGD, Adam, AdamW), learning rate (ranging from 1e-3 to 1e-5), loss functions (MSE, BCE, BCEWithLogitsLoss), batch size (32, 64, 128), and regularization methods (Dropout and Batch Normalization). These adjustments played a crucial role in both convergence and model robustness.

Automated Optimization with Optuna
In addition to manual trials, I employed the Optuna optimization framework to automatically search for the most effective hyperparameters. Optuna uses a sampling-based approach to efficiently explore the hyperparameter space and identify high-performing configurations. Specifically, the Optuna experiments focused on:

- Number of neurons in each hidden layer
- Learning rate
- Choice of optimizer (Adam vs. AdamW)

Due to computational cost (it runs in like 3 hours), the Optuna code is commented out in the final submission, but it can be activated and examined for evaluation purposes. The results obtained through Optuna confirmed the findings of the manual experiments and helped validate the final architecture and training settings.

3.4. Evaluation

To assess the effectiveness of the model, I evaluated its predictions using accuracy, precision, recall, and F1-score. These metrics provide a comprehensive understanding of the model's performance beyond just accuracy as it was showed in tutorial lesson.

Specifically:

- Accuracy: Measures the overall correctness of the model by calculating the ratio of correctly classified instances to the total instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

- Precision: Evaluates how many of the predicted positive instances were actually correct, helping to measure false positives.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

- Recall : Indicates how many of the actual positive instances were correctly identified, focusing on false negatives.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

- F1-score: The harmonic mean of precision and recall, providing a balanced measure.

$$F1Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{4}$$

Performance Results
The evaluation results on the validation dataset are shown in the table below:

	Precision	Recall	F1-score	Support
Class 0	0.79	0.79	0.79	21197
Class 1	0.79	0.79	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: Evaluation Metrics for the Final Model

These results confirm that the model performs consistently across both classes, with precision, recall, and F1-scores all at 0.79, indicating balanced performance and no class-specific bias. The evaluation also suggests that the preprocessing, embedding selection, and architecture tuning steps collectively led to a model capable of generalizing well across the data.

4. Results and Overall Analysis

4.1. Results Analysis

The final model achieved a validation accuracy of 0.79096 and a test accuracy of 0.78994 (on Kaggle), indicating strong and consistent generalization across unseen data. These results demonstrate that a well-tuned Deep Neural Network, paired with domain-specific GloVe Twitter embeddings, can effectively capture sentiment patterns in noisy, informal text such as tweets.

Evaluation Metrics on Validation Set

- Accuracy: 0.79003
- Precision: 0.79180
- Recall: 0.78702
- F1-Score: 0.78940

These evaluation metrics indicate that the model is both balanced and robust. The nearly identical values for precision (0.79180) and recall (0.78702) suggest that the classifier maintains a strong balance between false positives and false negatives. This means it is equally effective at correctly identifying both positive and negative sentiments, without disproportionately favoring one class over the other, a crucial trait for sentiment analysis tasks where both types of errors can be problematic. Moreover, the F1-score of 0.78940, which is the harmonic mean of precision and recall, provides a consolidated view of this trade-off. Its high value confirms that the model performs well not just in isolation for precision or recall, but in balancing the two, especially in borderline cases where the distinction between positive and negative sentiment is subtle.
Such consistent scores across all metrics reinforce the model's stability and reliability in real-world applications, where input data can be noisy, varied, and unstructured as is often the case with tweets. Importantly, the lack of skew toward either class indicates that the model has not overfit to one sentiment category. In summary, the performance of this model shows that the combination of effective preprocessing, suitable embeddings, and thoughtful model architecture has resulted in a well-calibrated sentiment classifier capable of handling noisy, real-world text.

Learning Curves

The figure below illustrates the learning curves of the final Deep Neural Network model, displaying the evolution of key training and validation metrics over its trainings epochs. Specifically, it plots:

- Training loss (blue) and training accuracy (orange)
- Validation loss (green) and validation accuracy (red)

These learning curves offer clear and valuable insights into the training dynamics and generalization performance of the model.

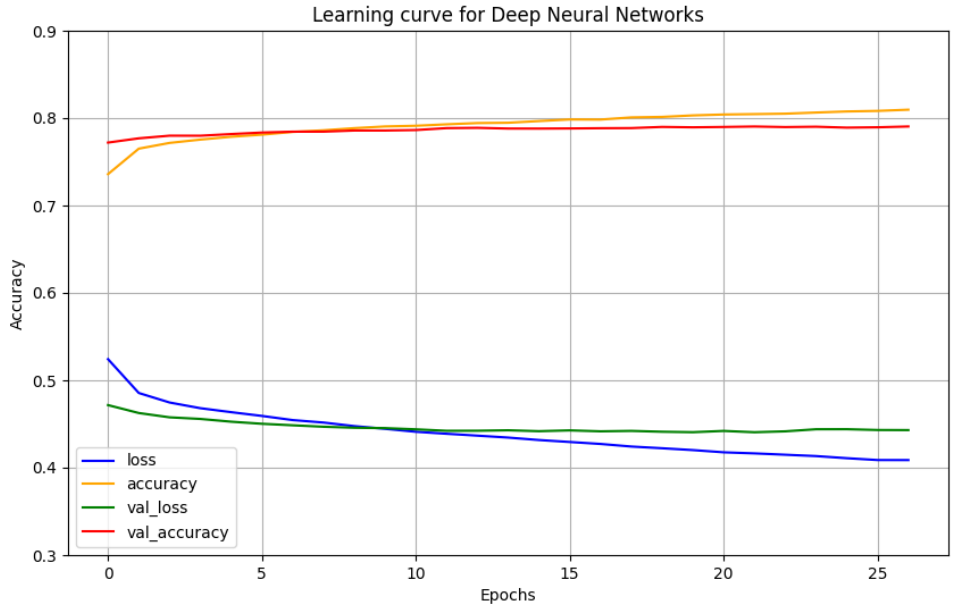

Figure 7: Final Model

The training loss exhibits a steady downward trend over the epochs, signifying that the model is consistently reducing its prediction error on the training set. This is a strong indicator that the optimization process is working effectively and that the model is learning meaningful patterns from the data.
In parallel, the training accuracy progressively increases, reinforcing the observation that the model is becoming more confident and correct in its predictions on known data. This coherence between decreasing loss and rising accuracy confirms stable and efficient learning.

The validation loss also shows a gradual decline before reaching a plateau, indicating that the model continues to improve on unseen data while avoiding excessive memorization of the training set. Its slower rate of decrease compared to training loss is expected and typical in well-regularized models.
The validation accuracy climbs quickly during the initial epochs and then levels off around epoch 15–20. Importantly, it stays consistently close to the training accuracy, with only a slight, stable gap between the two. This behavior suggests that the model has achieved good generalization: it performs nearly as well on new data as it does on the data it was trained on.

4.1.1. Best trial.

- Preprocessing:
  Extensive cleaning and normalization, including tokenization, contraction handling, removal of noise (mentions, hashtags, URLs), and spelling correction.
- Word Embeddings:
  Pretrained glove.twitter.27B.200d.txt embeddings, chosen for their domain specificity and semantic richness in Twitter-based language.

- Architecture:
  Deep Neural Network with 3 hidden layers of sizes 512, 256, and 128, activated with ReLU functions.
- Training Hyperparameters
  - Optimizer: Adam
  - Learning Rate: 1e-4
  - Loss Function: BCEWithLogitsLoss
  - Batch Size: 128
  - Epochs: 50 (with Early Stopping applied)
  - Dropout Rate: 0.3
  - Batch Normalization: applied before each activation

This trial balanced model complexity and regularization effectively, leading to both high accuracy and strong generalization performance. The consistent results across training, validation, and test sets demonstrate that the model architecture and hyperparameter choices were well-tuned for the task.

4.2. Comparison with the first project

In the first project, I implemented a sentiment classifier using Logistic Regression with TF-IDF feature extraction. The best-performing model in that setup achieved an accuracy of 0.80536 on the validation set and 0.80371 on the test set. In this second project, I built a Deep Neural Network (DNN) with GloVe Twitter embeddings and achieved a validation accuracy of 0.79096 and a test accuracy of 0.78994. At first glance, the Logistic Regression model outperforms the DNN slightly in terms of raw accuracy. However, a deeper comparison reveals important qualitative differences:

Why Logistic Regression performs slightly better:

- Simplicity of the task:
  Logistic Regression is a strong baseline for text classification, especially when paired with high-dimensional sparse features like TF-IDF. It benefits from the explicit presence of weighted n-grams (1-3), which capture direct phrase sentiment effectively.
- Faster optimization:
  Logistic Regression converges quickly and is less sensitive to hyperparameters, making it easier to fine-tune.
- TF-IDF coverage:
  The TF-IDF vectorizer with 50,000 features captures a wide and rich vocabulary from the training data, whereas GloVe embeddings are fixed and may lack vectors for rare or out-of-vocabulary words.

Why Deep Neural Networks are still important

- Semantic understanding:
  Unlike TF-IDF, which treats text as a bag of words, the DNN with GloVe embeddings captures semantic relationships between words. This is especially useful for understanding contextual sentiment.

- Scalability and Extensibility:
  The Deep Neural Network model offers a more flexible and scalable foundation for future developments compared to Logistic Regression. While logistic models are limited to linear decision boundaries and fixed input features, neural networks can be easily extended to incorporate more advanced architectures such as transformers, attention mechanisms, or transfer learning pipelines. Even if Logistic Regression performed slightly better on this specific task, the DNN provides a more future-proof solution, ready to scale to more complex NLP tasks or larger, more diverse datasets.

In conclusion, while the Logistic Regression model slightly outperformed the DNN in terms of accuracy, the deep learning model offers greater semantic depth, flexibility, and long-term scalability. Given more data or a more complex sentiment task, the DNN would likely outperform the simpler model.

5. Bibliography

References

[1] Manolis Koubarakis. Perceptrons and backpropagation. 2025.

[2] Manolis Koubarakis. Training dnns, word vectors and the word2vec model. 2025.

[3] PyTorch.org. torch.nn, https://pytorch.org/docs/stable/nn.html.

[4] Ahmed Yassin. Understanding the difference between model.eval() and model.train() in pytorch, https://yassin01.medium.com/understanding-the-difference-between-model-eval-and-model-train-in-pytorch-48e3002ee0a2.

[1] [2] [3] [4]