Multi-Class Text Classification using CNN-LSTM and Transformer-based Architectures

Part A: CNN-LSTM Architecture

1 Introduction

Text classification is an important task in natural language processing with applications in spam detection, sentiment analysis, and topic categorization. In this work, we focus on multi-class classification for texts belonging to five categories: business, tech, politics, sport, and entertainment. Our proposed approach integrates CNN and LSTM components to exploit both local feature extraction and sequential context, which is essential for understanding the nuances of natural language.

2 Model Architecture

The chosen CNN-LSTM architecture is designed to capture complementary information from the text:

- **CNN Component:** Convolutional layers are used to extract local features (such as n-gram patterns) from word embeddings. The pooling operation then retains the most significant features, which helps in detecting salient phrases regardless of their position in the text.
- **LSTM Component:** The LSTM layer processes the sequence of embeddings to capture long-term dependencies and contextual relationships between words. Its bidirectional implementation allows the model to consider both past and future contexts.

The concatenation of outputs from the CNN and LSTM branches provides a rich representation that combines both local and global contextual information, resulting in improved classification performance. This design choice is further validated by our high accuracy and F1 scores on the test set.

3 Pipeline Overview

3.1 Data Preprocessing

- **Tokenization:** The texts are tokenized using NLTK's `word_tokenize` function.
- **Vocabulary Construction:** A vocabulary is built based on token frequency, with a minimum threshold to filter out rare words. Special tokens such as `<pad>` and `<unk>` are used for padding and unknown words.
- **Dataset Splitting:** The training data is split into training (80%) and validation (20%) sets.

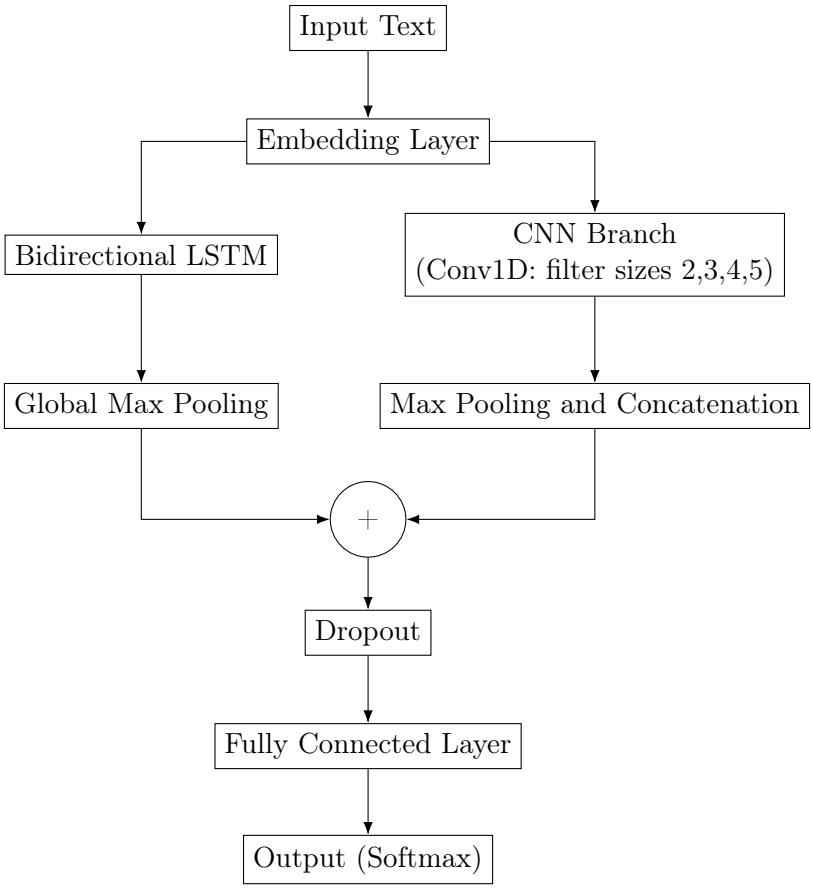

Figure 1: Model Architecture: The input text is embedded and then processed by two parallel branches: a Bidirectional LSTM (with Global Max Pooling) on the left, and a CNN (with multiple Conv1D filters followed by Max Pooling and Concatenation) on the right. Their outputs are merged and then passed through Dropout, a Fully Connected layer, and finally a Softmax output.

3.2 Model Training and Evaluation

- **Training:** The model is trained using the Adam optimizer with weight decay, and a learning rate scheduler that reduces the rate on a plateau. Gradient clipping is applied to prevent exploding gradients.
- **Validation:** The model's performance is monitored on the validation set after each epoch. The best performing model is saved based on validation accuracy.
- **Testing:** The best model is evaluated on the test set using metrics such as accuracy, precision, recall, and F1 score.

4 Results

- **Train Loss:** 0.0192, **Train Accuracy:** 99.41%
- **Validation Loss:** 0.0725, **Validation Accuracy:** 97.65%
- **Test Accuracy:** 94.83%

- **Test Precision:** 94.82%
- **Test Recall:** 94.85%
- **Test F1 Score:** 94.83%

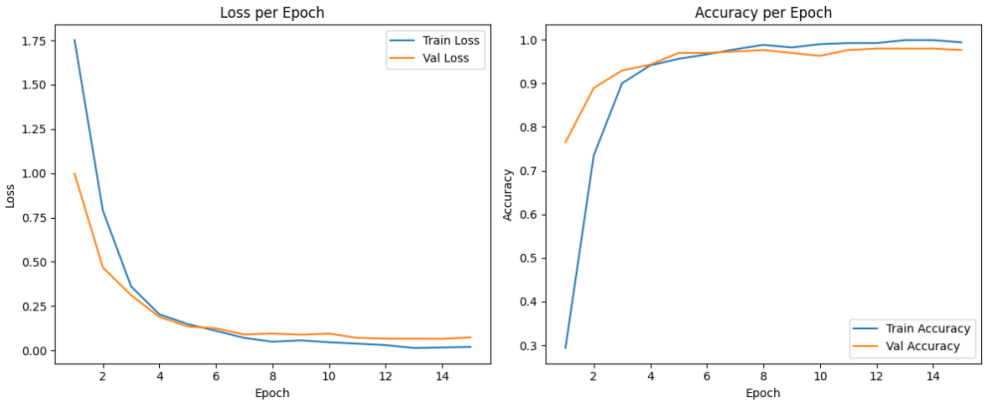

Figure 2: Training and validation loss and accuracy over 15 epochs.

5 Experiments

5.1 Tokenizer Experiments

We compared three tokenization strategies—NLTK, Whitespace, and a custom Regex tokenizer—keeping all other settings identical. Table 1 summarizes their performance on the test set.

Tokenizer	Accuracy	Precision	Recall	F1 Score
NLTK (`word_tokenize`)	94.97%	95.22%	94.78%	94.97%
Whitespace	94.56%	94.85%	94.52%	94.63%
Regex (custom pattern)	**96.33%**	**96.31%**	**96.37%**	**96.33%**

Table 1: Test metrics for different tokenizers.

The Regex tokenizer outperforms the others, achieving a 96.33% F1 score. We believe this is because:

- **Regex Tokenizer:** By explicitly splitting on punctuation, URLs, and special characters, it yields cleaner, more consistent tokens. This reduces noise (e.g., stray punctuation attached to words) and helps the model learn more robust n-gram features.
- **NLTK Tokenizer:** Provides linguistically informed splits (handling contractions, punctuation, etc.), but can be over-sensitive—splitting off apostrophes or hyphens in ways that increase vocabulary size and sparsity.
- **Whitespace Tokenizer:** Fast and simple, but lumps punctuation onto words (e.g., "hello," stays as "hello,"), which again increases noise and harms the embedding quality.

Pros and Cons

Regex Pros: Highly customizable; captures domain-specific patterns (URLs, emoticons, hashtags). **Cons:** Requires manual pattern design; may miss edge cases.

NLTK Pros: Off-the-shelf, well-tested; handles many languages. **Cons:** Slower; can over-split and increase sparsity.

Whitespace Pros: Fastest; zero dependencies. **Cons:** Ignores punctuation boundaries; lower accuracy.

Based on these results, we adopt the Regex tokenizer for all subsequent experiments.

5.2 Embedding Experiments

We evaluated multiple embedding strategies for the text classification task: three pretrained embeddings—GloVe, Word2Vec, and BERT; a meta-embedding that combines GloVe and Word2Vec; and an *Original Embedding* learned from scratch during training. All experiments used the same Regex tokenizer and model architecture.

Embedding Descriptions

- **GloVe:** A global co-occurrence-based embedding trained on massive corpora (e.g., Common Crawl). Each word has a fixed vector, offering semantic regularity with low computational cost.
- **Word2Vec:** Predicts word neighborhoods using local context (CBOW or Skip-Gram). While effective in capturing word associations, it generates static vectors and struggles with rare or ambiguous words.
- **BERT:** A deep Transformer-based model producing context-aware embeddings. Each token's representation depends on its surrounding words, enabling better handling of polysemy.
- **Meta-Embedding:** Concatenates GloVe and Word2Vec embeddings and uses a gating mechanism to learn a weighted combination. It leverages complementary information from both static embedding spaces.
- **Original Embedding:** Randomly initialized and trained from scratch along with the model. It adapts fully to the downstream task but lacks pretrained linguistic knowledge.

Embedding	Accuracy	Precision	Recall	F1 Score
GloVe (pretrained)	96.33%	96.29%	96.58%	96.42%
Word2Vec (pretrained)	87.89%	87.97%	87.64%	87.50%
BERT (pretrained)	96.33%	96.26%	96.70%	96.40%
Meta-Embedding (pretrained)	**96.60%**	**96.56%**	**96.75%**	**96.64%**
Original Embedding	96.33%	96.31%	96.37%	96.33%

Table 2: Test metrics for different embedding strategies using the Regex tokenizer. All except "Original Embedding" are pretrained.

Analysis The meta-embedding delivers the best performance across all metrics, indicating that combining the global semantics of GloVe with the local syntactic relationships of Word2Vec results in a more expressive and generalizable representation.

The *Original Embedding*, despite being trained from scratch without any external knowledge, performs competitively—outperforming Word2Vec and closely trailing GloVe and BERT. This suggests that with sufficient training data and a strong architecture, task-specific embeddings can be highly effective.

BERT and GloVe show nearly identical results, with BERT having a slight edge in recall and F1, likely due to its contextual modeling. However, BERT is more resource-intensive. Word2Vec, on the other hand, consistently underperforms, likely due to its inability to model context or disambiguate meaning in complex input sequences.

Pros and Cons

GloVe: Pros: Efficient; captures global co-occurrence. **Cons:** Lacks context-awareness; static word vectors.

Word2Vec: Pros: Captures syntactic proximity. **Cons:** Underperforms; poor handling of rare/ambiguous words.

BERT: Pros: Context-sensitive; handles polysemy. **Cons:** Computationally heavy; slower inference.

Meta-Embedding: Pros: Combines complementary strengths; best performance. **Cons:** Larger embedding size; more memory usage.

Original Embedding: Pros: Tailored to the task; no external dependencies. **Cons:** Needs sufficient training data; longer convergence.

5.2.1 Dynamic Meta-Embedding Experiments

In this experiment, we replaced the single embedding layer with *dynamic meta-embeddings*: two trainable embedding matrices combined by a learned gating network, all trained end-to-end. The test metrics are shown in Table 3.

Embedding Type	Accuracy	Precision	Recall	F1 Score
Original Single Embedding	96.33%	96.31%	96.37%	96.33%
Dynamic Meta-Embedding (trained)	95.92%	95.80%	96.09%	95.92%

Table 3: Test metrics for dynamic meta-embeddings vs. the original single embedding.

Interpretation Although dynamic meta-embeddings can in principle learn complementary representations, we observe a slight drop (0.4 % in F1) compared to the single embedding baseline. Possible reasons:

- **Increased model complexity:** The gating network plus two embeddings adds many parameters, which can overfit or require more data to train effectively.
- **Training dynamics:** Jointly learning both embeddings and the gate may lead to suboptimal gating early on, limiting the benefit of the second embedding.

- **Representation redundancy:** When both embeddings are trained from scratch on the same data, they may not diverge enough to offer truly complementary signals.

5.3 CNN Module Experiments

To further improve performance, we conducted targeted experiments on the CNN module of our architecture, focusing on two key design choices: the number of filters and the kernel size configurations. Each experiment used the same dataset split, tokenizer (Regex), and Orginal Embedding strategy.

5.3.1 Experiment 1: Varying Number of Filters

We tested three different settings for the number of convolutional filters: 100, 150, and 200. The kernel configuration was fixed to $[2, 3, 4, 5]$ for all settings.

Filters	Test Accuracy	F1 Score
100	94.97%	94.87%
150	**96.33%**	**96.33%**
200	95.65%	95.61%

Table 4: Performance with different numbers of CNN filters.

Interpretation The results indicate that increasing the number of filters from 100 to 150 leads to a noticeable improvement in both accuracy and F1 score. However, increasing to 200 filters yields only a marginal gain in F1 but slightly lower accuracy, possibly due to overfitting or diminishing returns. Thus, 150 filters offer the best balance between model complexity and generalization.

5.3.2 Experiment 2: Kernel Size Configurations

We evaluated several kernel size setups to assess their impact on capturing n-gram patterns of different lengths. The number of filters per kernel was 150.

Kernel Sizes	Test Accuracy	F1 Score
$[3, 3, 3]$	95.78%	95.76%
$[3, 4, 5]$	95.24%	95.22%
$[2, 3, 4, 5]$	**96.33**%	**96.33**%
$[11, 11, 11]$	91.56%	91.31%

Table 5: Performance under different kernel size configurations.

Interpretation The best performance was achieved with the kernel size configuration [2, 3, 4, 5], which includes a wider variety of n-gram filters. This setup resulted in the highest test accuracy (96.33%) and F1 score (96.33%), indicating that combining multiple kernel sizes allows the model to capture a richer set of local patterns from short to slightly longer phrases. In contrast, the configuration with only wide kernels [11,11,11] performed the worst, likely due to over-smoothing and the loss of fine-grained local features. Interestingly, the setup with repeated tri-gram filters

[3, 3, 3] still performed competitively, highlighting the effectiveness of tri-gram modeling but also suggesting that added diversity in kernel sizes boosts performance.

5.3.3 Best CNN Configuration

Combining insights from both experiments, the optimal CNN setup uses:

- **Number of Filters:** 150
- **Kernel Sizes:** $[2, 3, 4, 5]$

This configuration achieved a test accuracy of **96.33%** and an F1 score of **96.33%**, outperforming all other tested setups. These findings will guide our model architecture for future experiments.

5.4 LSTM Module Experiments

We next evaluated the recurrent component by comparing unidirectional vs. bidirectional LSTM, varying the number of layers and hidden dimensions, and also including a GRU baseline. Table 6 summarizes test metrics.

RNN	Bi-Dir?	Layers	Hidden	Acc.	Prec.	Rec.	F1
LSTM	Yes	1	128	95.10%	95.08%	95.13%	95.10%
LSTM	No	1	128	95.37%	95.48%	95.31%	95.36%
LSTM	Yes	1	256	96.33%	96.24%	96.46%	96.33%
LSTM	No	1	256	**96.46%**	**96.62%**	**96.52%**	**96.57%**
LSTM	Yes	1	512	93.33%	93.07%	93.60%	93.12%
LSTM	No	1	512	95.51%	95.42%	95.55%	95.47%
LSTM	Yes	2	128	95.78%	95.68%	95.87%	95.77%
LSTM	No	2	128	93.47%	93.14%	93.79%	93.26%
LSTM	Yes	2	256	94.15%	94.16%	93.87%	93.97%
LSTM	No	2	256	93.47%	93.93%	93.18%	93.46%
LSTM	Yes	2	512	93.61%	93.65%	93.51%	93.57%
LSTM	No	2	512	95.10%	95.10%	95.17%	95.13%
GRU	–	1	128	94.83%	94.70%	94.80%	94.74%
GRU	–	1	256	95.24%	95.20%	95.37%	95.24%
GRU	–	1	512	95.92%	95.93%	95.99%	95.93%
GRU	–	2	128	94.42%	94.24%	94.52%	94.33%
GRU	–	2	256	94.15%	94.09%	94.27%	94.16%
GRU	–	2	512	92.93%	92.87%	92.97%	92.85%

Table 6: Test performance for LSTM (uni- vs. bi-directional), varying layers and hidden size, plus GRU baselines.

Interpretation

- **Best overall:** A single-layer, *unidirectional* LSTM with 256 hidden units achieved the highest F1 (96.57%) and accuracy (96.46%).

- **Bidirectional vs. Unidirectional:** For 1-layer LSTMs, unidirectional slightly outperforms bidirectional (e.g. 95.36% vs. 95.10% F1 at 128 units; 96.57% vs. 96.33% at 256 units), suggesting that the extra parameters in the backward pass can sometimes overfit on this dataset.
- **Depth:** Adding a second LSTM layer generally hurt performance (e.g. 2-layer 128 hidden: 95.77% vs. 95.36% F1 for 1-layer), indicating that deeper recurrence did not pay off given our data size.
- **Hidden size:** Very large hidden sizes (512) led to overfitting in bidirectional LSTMs (only 93.12% F1), whereas a unidirectional 512-unit LSTM still held up at 95.47%.
- **GRU baseline:** A 1-layer GRU with 512 units was the strongest GRU model (95.93% F1), but still fell short of the best LSTM. GRUs with more depth or smaller hidden sizes underperformed compared to their LSTM counterparts.

5.5 Attention at LSTM Output

In this additional experiment, we replaced the global max-pooling over the bidirectional LSTM outputs with a learned attention mechanism. The model was otherwise identical to our best CNN–LSTM setup (150 CNN filters, kernel sizes [2,3,4,5], single-layer bidirectional LSTM with 256 units). The test metrics are shown in Table 7.

Pooling/Attention	**Accuracy**	**Precision**	**Recall**	**F1 Score**
Global Max-Pooling (baseline)	96.33%	96.24%	96.46%	96.33%
Attention at LSTM Output	94.56%	94.34%	94.68%	94.46%

Table 7: Test performance when using attention instead of global max-pooling on the LSTM branch.

Interpretation Although attention mechanisms often yield richer, context-aware representations, here it led to a roughly 2 % drop in F1 compared to max-pooling. We hypothesize that:

- **Over-parameterization:** The attention layer adds extra weights and non-linearities, which may overfit given our dataset size.
- **Signal Dilution:** Max-pooling explicitly selects the strongest activation (the most salient feature), whereas attention distributes weight across many time-steps, potentially diluting the peak signals that are most predictive for classification.
- **Optimization Difficulty:** Training the attention weights jointly with the rest of the network can be more sensitive to hyperparameters and may converge to suboptimal alignments without additional regularization or a larger dataset.

These findings suggest that, for our five-way topic classification task and data scale, the simplicity and hard-selection bias of global max-pooling better preserves the most discriminative features than a soft-attention mechanism.

6 Conclusion

We have presented a hybrid CNN–LSTM model for five-way text classification and conducted extensive experiments on tokenization, embeddings, CNN configurations, recurrent modules, attention, and dynamic meta-embeddings. Our key takeaways are:

- A custom regex tokenizer yields the cleanest tokens and drives the strongest performance (96.33% F1).
- Meta-embeddings (GloVe + Word2Vec with learned gating) outperform single pretrained embeddings, achieving up to 96.64% F1.
- The optimal CNN branch uses 150 filters with kernel sizes [2,3,4,5], yielding 96.33% F1.
- In the recurrent branch, a single-layer *unidirectional* LSTM with 256 hidden units achieved the best results (96.57% F1, 96.46% accuracy).
- Replacing global max-pooling with a soft attention mechanism on the LSTM outputs degraded performance by 2% F1 (from 96.33% to 94.46%), likely due to over-parameterization, signal dilution, and optimization challenges.
- Dynamic meta-embeddings trained from scratch led to a slight drop (0.4% F1; 95.92% vs. 96.33%), likely due to increased parameter count, training complexity, and redundancy in learned representations.

These findings underscore the power of simplicity—hard-selection pooling and a single embedding layer can outperform more complex mechanisms on moderate-sized classification tasks.

7 Future Work

Building on our CNN–LSTM findings and the new attention experiment, we plan to explore the following directions:

- **Refined Attention Mechanisms:** Investigate alternative attention formulations (e.g. multi-head, self-attention) and regularization techniques (dropout on attention weights, sparsity constraints) to see if they can outperform global pooling without overfitting.
- **Hybrid Pooling–Attention:** Combine hard-selection (max-pooling) with soft-attention (e.g. gated pooling, top-k attention) to retain the strongest signals while still leveraging context weighting.
- **Scale and Data Augmentation:** Increase training data via weak supervision or data augmentation (back-translation, synonym replacement) to give attention mechanisms more examples and reduce over-parameterization risks.
- **Transformer Hybrids:** Replace the LSTM branch with lightweight Transformer blocks (e.g. ALBERT-style) that natively integrate multi-head attention, and compare against the current CNN–LSTM baseline.
- **End-to-End Pretrained Models:** Fine-tune full Transformer encoders (BERT, RoBERTa) in place of the LSTM branch, with and without additional attention pooling, to benchmark against our custom architectures.

Part B: Transformer-based Architecture

8 Introduction

This report presents a comprehensive pipeline for multi-class text classification using a Transformer-based model. The dataset consists of text samples labeled with five categories: `business`, `tech`, `politics`, `sport`, and `entertainment`. The objective is to accurately classify the input text into one of these predefined categories using a deep learning architecture.

9 Model Architecture

The Transformer-based architecture is designed to capture both global contextual information and token-level dependencies in the text:

- **Embedding and Positional Encoding:** Tokens are converted into dense vectors using a learned embedding layer. Positional encoding is added to retain word order information, which is not inherently preserved by the attention mechanism.
- **Transformer Encoder Layers:** The model includes multiple stacked encoder layers, each consisting of multi-head self-attention and position-wise feedforward sublayers. Residual connections and layer normalization stabilize training and improve gradient flow.
- **Pooling and Classification:** The outputs from the encoder are aggregated using mean pooling, followed by a fully connected layer to produce class logits. Softmax is applied to obtain final class probabilities.

This architecture enables the model to focus on relevant parts of the input text using self-attention, while also preserving sequential structure through positional encoding. Its performance is validated by strong evaluation metrics on the test set. The inner structure of the transformer encoder block is shown in Figure 3.

10 Pipeline Overview

10.1 Data Preprocessing

- **Tokenization:** A custom regex-based tokenizer is used to extract word-level tokens from the text.
- **Vocabulary Construction:** A vocabulary is built using the training set, applying a minimum frequency threshold to remove infrequent tokens. Special tokens like `<pad>` and `<unk>` are included for padding and unknown words.
- **Dataset Splitting:** The dataset is divided into training (80%) and validation (20%) sets.

Figure 4 illustrates the complete pipeline from input text to final class prediction.

10.2 Model Training and Evaluation

- **Training:** The model is trained using the AdamW optimizer along with a cross-entropy loss function. A ReduceLROnPlateau scheduler is used to adjust the learning rate dynamically based on validation accuracy. Gradient clipping is employed to prevent exploding gradients.

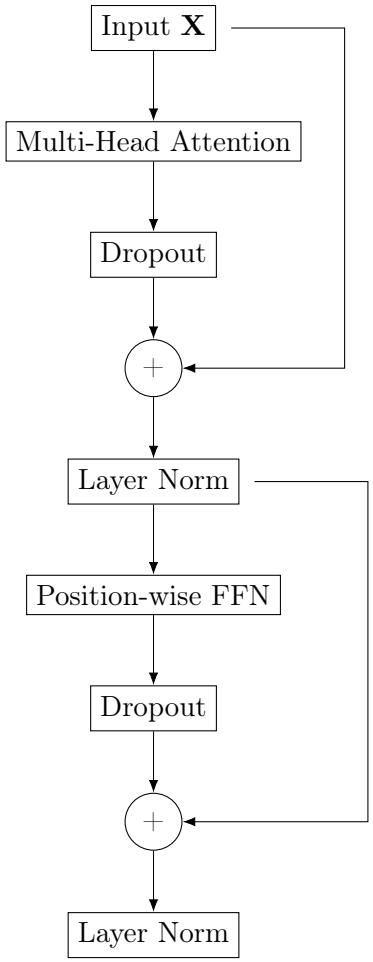

Figure 3: Architecture of a Transformer Encoder Block.

- **Validation:** Performance is evaluated on the validation set after each epoch. The best-performing model is selected based on validation accuracy and saved for testing.
- **Testing:** The final evaluation is performed on the held-out test set using accuracy, precision, recall, and F1 score (macro-averaged).

11 Results

- **Train Loss:** 0.0275, **Train Accuracy:** 98.83%
- **Validation Loss:** 0.0591, **Validation Accuracy:** 96.80%
- **Test Accuracy:** 93.06%
- **Test Precision:** 93.00%
- **Test Recall:** 93.22%
- **Test F1 Score:** 93.06%

Figure 5 shows training and validation loss and accuracy over epochs.

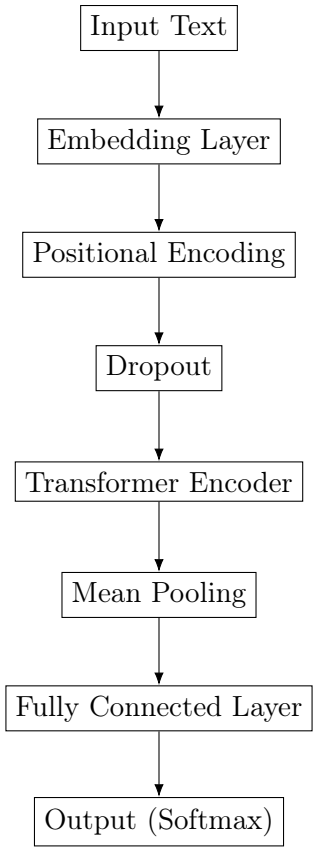

Figure 4: Overall Transformer-based model pipeline for text classification.

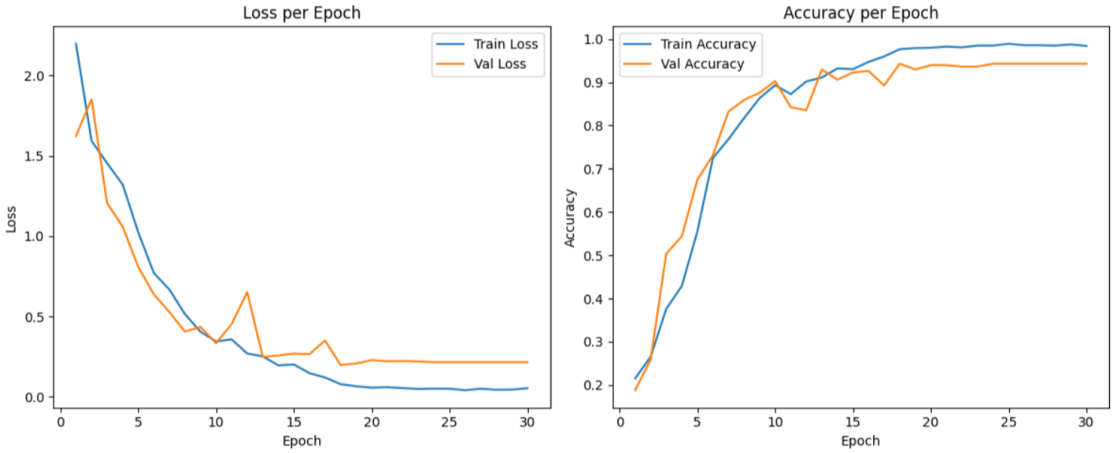

Figure 5: Training and validation loss and accuracy per epoch

12 Experiments

This section presents a series of experiments to evaluate and refine the proposed Transformer-based model for multi-class text classification. Each experiment modifies one component while keeping others fixed to isolate the effect of that change.

12.1 Experiment 1: Positional Encoding Strategies

To assess the role of positional encoding, we experimented with:

- **Learnable Positional Encoding**: Initialized randomly and learned during training.
- **No Positional Encoding**: Removes explicit positional information.

Configuration	Accuracy	Precision	Recall	F1 Score
Learnable Positional Encoding	92.93%	92.80%	93.04%	92.74%
No Positional Encoding	**95.37%**	**95.19%**	**95.52%**	**95.32%**

Table 8: Comparison of positional encoding strategies.

Interpretation Surprisingly, the model performed better without any positional encoding. This may indicate that token position is less relevant for this dataset, or that the model implicitly learns position from context.

12.2 Experiment 2: Using Max Pooling

Instead of Mean Pooling we applied Max Pooling to the tokens output from the transformer encoder block

Representation Method	Accuracy	Precision	Recall	F1 Score
Mean Pooling	**95.37**%	**95.19**%	**95.52**%	**95.32**%
Max Pooling	94.54%	94.54%	94.54%	94.54%

Table 9: Effect of different sequence representation strategies.

Interpretation Using the Max Pooling token degraded performance. This suggests that for our task and architecture, aggregating information via mean pooling over all token embeddings is more effective.

12.3 Experiment 3: Removing Positional Encoding + Adding `[CLS]` Token

We also explored removing positional encoding and using a `[CLS]` token for classification.

Configuration	Accuracy	Precision	Recall	F1 Score
No Pos. Enc. + `[CLS]` Token	93.47%	93.47%	93.47%	93.39%

Table 10: Performance of using `[CLS]` token without positional encoding.

Interpretation The performance remained lower than the baseline (mean pooling + no positional encoding), reaffirming that mean pooling is preferable and `[CLS]`-based summarization does not help here.

12.4 Experiment 4: Varying Number of Heads and Encoder Layers without Positional Encoding

This experiment investigates how the number of attention heads and stacked encoder layers affects performance. The following configurations were evaluated:

Configuration (Heads, Layers)	Accuracy	Precision	Recall	F1 Score
(4, 4)	93.61%	93.68%	93.54%	93.59%
(4, 6)	94.29%	94.55%	94.20%	94.31%
(4, 8)	93.47%	93.26%	93.68%	93.36%
(8, 4)	93.47%	93.26%	93.68%	93.36%
(8, 6)	**95.37%**	**95.19%**	**95.52%**	**95.32%**
(8, 8)	92.65%	92.47%	92.75%	92.37%

Table 11: Performance of Transformer with different numbers of attention heads and encoder layers.

Interpretation The model achieved the best performance when using **8 heads and 6 layers**, suggesting that increasing attention granularity and moderate depth contributes to better contextual modeling. However, beyond a certain depth (e.g., 8 layers), performance declined, indicating possible overfitting or optimization difficulty. This experiment confirms that tuning the number of heads and layers is crucial to achieving optimal performance.

12.5 Comparison of Transformer Configurations

To summarize the key observations, Table 12 compares all major configurations tested.

Configuration	Accuracy	Precision	Recall	F1 Score
Learnable Positional Encoding	92.93%	92.80%	93.04%	92.74%
No Positional Encoding	**95.37%**	**95.19%**	**95.52%**	**95.32%**
No Pos. Enc. + `[CLS]` Token	93.47%	93.47%	93.47%	93.39%

Table 12: Summary of Transformer architecture variations and their performance.

13 Conclusion

The key findings from our Transformer-based multi-class text classification experiments are summarized below:

- **No positional encoding** surprisingly outperformed models with learnable positional embeddings, suggesting the dataset structure allows the model to infer positional relationships implicitly.
- **Mean pooling** over token embeddings yielded better performance than using a dedicated `[CLS]` token, contrary to trends observed in pretrained language models.
- **The combination** of removing positional encoding and introducing a `[CLS]` token did not improve performance, reinforcing the strength of mean pooling.

- **Optimal configuration** for attention heads and encoder layers was found to be 8 heads and 6 layers, balancing model capacity with generalization.

Overall, careful architectural choices—especially in sequence representation and encoder depth—play a crucial role in achieving optimal performance on this classification task.

14 Future Work

Several directions remain open to further improve and understand Transformer-based models in this context:

- **Investigate learned token importance** through attention visualization to interpret model decisions more transparently.
- **Experiment with pre-trained Transformer embeddings** such as BERT or RoBERTa in a fine-tuning setup for potential performance gains.
- **Incorporate task-specific inductive biases** through relative or rotary positional encodings instead of removing position entirely.
- **Apply label smoothing or focal loss** to handle class imbalance more effectively, especially for low-frequency categories.
- **Conduct ablations on input embedding strategies**, including dynamic meta-embeddings and domain-adaptive pretraining.