INFO08010 Project: 7ess-Xin
Sentiment analysis with Transformers

Guenfoudi Ihabe[1] and **Liu Eléna**[2]

[1] *Ihabe.Guenfoudi@student.uliege.be (s194071)*
[2] *elena.liu@student.uliege.be (s201772)*

In this paper, we introduce *7ess-Xin*, a lightweight encoder-only Transformer trained from scratch on the Sentiment140 dataset for Twitter sentiment classification, formulating the task as binary sequence classification. We detail our end-to-end pipeline, including comprehensive tweet cleaning and normalization, byte-level Byte-Pair Encoding subword tokenization, and the design of ten custom Transformer variants. We systematically compare sinusoidal versus Rotary Positional Embeddings and the addition of Query–Key normalization, demonstrating that these enhancements yield consistent improvements without increasing parameter count. Our best model achieves an F1-score of 83.25% and an accuracy of 83.13% on a held-out test set. We also analyze evaluation challenges, model biases toward positive sentiment, and the impact of noisy labels in the dataset. The full implementation and examples are publicly available at https://github.com/GuenfoudiIhabe/7ess-Xin.

I. INTRODUCTION AND MOTIVATION

Emotions and sentiments are complex and fundamental aspects of human nature. They do not only reflect an individual's internal state, but also influence cognition, behavior, and decision making [1]. Based on this observation, our project aims to predict the emotional state of individuals through textual data, particularly from social media, as a basis for understanding and anticipating potential behavioral trends.

We named our project *7ess-Xin*, short for "heart sentiments" where 7ess (Arabic) refers to feeling or emotion, and Xin (Chinese) means heart or mind. It focuses on the Sentiment140 dataset, which contains 1.6 million labeled tweets. The main objective of our work is to classify whether a user's emotion, at the time of tweeting, was positive or negative. Our key research question is: Can a lightweight encoder-only Transformer trained from scratch on Sentiment140 match the 80% ceiling without external pre-training? See Subsection V B for an analysis of the factors underlying this performance ceiling.[2]

Although many related works have addressed this task (Section II), we chose to design and train a custom transformer-based model. In this report, we explain the methodology followed, including the architecture of our neural network model, the hyperparameters used, the loss function, and a detailed description of the data and its preprocessing (Section III). We then present the results of our experiments through both quantitative and qualitative analyses using the Weights & Biases tracking platform [3] (Section IV). Finally, we conclude with an overall evaluation of the model's performance, a discussion of its limitations, and possible future improvements (Section V).

A. Work Distribution

This project was originally planned for a team of three students, but one team member (Clément) abandoned the project after the proposal. The remaining work was distributed as follows:

- **Ihabe Guenfoudi:** Primary code implementation, model architecture design, experiment running, and report review.
- **Elena Liu:** Literature research, paper review, report writing, and experimental results analysis.

II. RELATED WORK

Several Transformer-based models for sentiment classification introduce architectural and training innovations that enhance accuracy, generalization, or efficiency [4]. We highlight a few variants most relevant to our work.

BERT remains a widely used benchmark, offering strong performance on sentiment analysis tasks through masked language modeling pretraining [5]. Building on BERT, RoBERTa (A Robustly Optimized BERT Pretraining Approach) removes the Next Sentence Prediction objective, applies dynamic masking, and trains on larger corpora with bigger batch sizes to improve text understanding [6]. DistilBERT creates a smaller, faster model via knowledge distillation, where a student model learns to mimic a pretrained BERT "teacher," achieving competitive accuracy with lower computational cost—ideal for real-time or resource-constrained settings [7]. ALBERT (A Lite BERT) reduces parameter count through factorized embedding parameterization and cross-layer weight sharing, maintaining high performance while cutting memory and training time [8]. T5 (Text-to-Text Transfer Transformer) reframes all NLP tasks as text-to-text generation problems, enabling a single unified architecture across diverse tasks [9].

Unlike these large, pretrained giants, our work investigates how far a lightweight encoder-only Transformer—trained *from scratch* on the 1.6 million-tweet Sentiment140 dataset—can go without any external pretraining.

III. METHODOLOGY

In this section, we describe the full methodological pipeline: starting with the dataset and preprocessing steps, followed by the architecture of the Transformer-based model, the training hyperparameters, and the evaluation metrics, including the loss function used during optimization.

A. Data and Preprocessing

We load the full Sentiment140 dataset [10], which contains 1.6 million tweets automatically labeled as positive (4) or negative (0). The `load_sentiment140` function immediately remaps these labels to 0 and 1, respectively, so our task reduces to binary sequence classification: the model reads each tweet and predicts one of two classes.

1. *Preprocessing*

To help the model focus on meaningful linguistic patterns and avoid overfitting to noise, we apply the following cleaning steps in `normalize_tweet_text`:

- **Lowercasing:** Convert all characters to lowercase, so "Happy" and "happy" are treated the same and our vocabulary stays manageable.
- **URL removal:** Strip out all substrings matching `http://`, `https://` or `www.` links.
- **Handle removal:** Remove Twitter handles of the form `@username`.
- **Hashtag handling:** Delete the '#' symbol but keep the tag text (e.g. `#excited` → `excited`), since the word often carries sentiment information.
- **Punctuation and digit removal:** Drop all non-alphanumeric characters and digits, which rarely contribute to sentiment and can introduce noise.
- **Whitespace normalization:** Collapse any run of whitespace into a single space and trim leading/trailing spaces to ensure clean token boundaries.

The cleaned text is stored in a new column `normalized_text`, leaving the raw tweet in `content` for reference.

2. *Tokenization*

We train a byte-level Byte-Pair Encoding (BPE) tokenizer on the entire `normalized_text` corpus, using a vocabulary of 30 000 subword tokens [11]. We reserve four special tokens:

- `<pad>` (ID 0) for padding
- `<s>` (ID 1) as the start-of-sequence marker
- `</s>` (ID 2) as the end-of-sequence marker
- `<unk>` (ID 3) for any out-of-vocabulary token

Our tokenizer uses Hugging Face's ByteLevel pre-tokenizer (adding a leading space) and ByteLevel decoder, making it robust to informal Twitter text—slang, typos, emojis, and abbreviations.

3. *Padding and Truncation*

During data batching, each tweet's token ID list is handled as follows:

- If the token list exceeds our fixed maximum length of 128, it is truncated to the first 128 IDs.
- If it is shorter than 128, we pad on the right with the `<pad>` token (ID 0) until length 128.

This ensures that every input tensor has shape `[batch_size, 128]`, allowing efficient parallel processing on the GPU.

4. *Dataset Splitting*

We partition the cleaned dataset via stratified sampling on the `sentiment` label to preserve the positive/negative ratio in each split. Specifically, we first reserve 20% of the data using Scikit-learn's `StratifiedShuffleSplit` (with `test_size`=0.2). We then split that 20% evenly into validation and test sets, resulting in:

- **Learning Set (LS):** 80% of the data for training
- **Validation Set (VS):** 10% of the data for hyperparameter tuning and early stopping
- **Test Set (TS):** 10% of the data for final performance evaluation

This 80%/10%/10% stratified split guards against class-imbalance bias during training and evaluation.

B. Model Architecture

Our model is an *encoder-only Transformer* tailored to sequence classification. Unlike the original encoder–decoder Transformer of Vaswani *et al.* [12], we omit the decoder—mirroring BERT's encoder stack—because we classify tweets rather than generate text.

Figure 1 shows the overall schema; the components are detailed below.

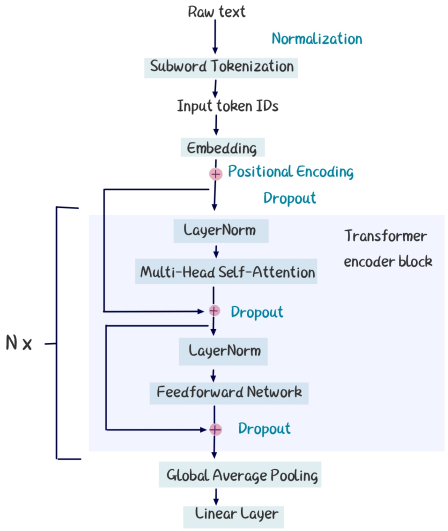

FIG. 1: Transformer encoder used in this work. After token and positional embeddings, the sequence passes through L encoder blocks (multi-head attention + FFN), then global mean pooling and a linear classifier. Hyperparameters appear in Table I.

1. *Embedding Layer*

A sequence of BPE token IDs $\langle w_1, \ldots, w_n \rangle$ is mapped to vectors via

$$\mathbf{X}_{\text{emb}} = \mathbf{E}\big(w_{1:n}\big), \qquad \mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}},$$

where $d_{\text{model}} \in \{128, 192, 256\}$ in our experiments. The embeddings are summed with positional encodings and passed through an input-dropout layer (p=0.1).

2. *Sinusoidal Positional Encoding*

To encode word order we add fixed, non-learned sinusoids [12]:

$$PE_{t,2i} = \sin\big(t/10000^{2i/d_{\text{model}}}\big),$$
$$PE_{t,2i+1} = \cos\big(t/10000^{2i/d_{\text{model}}}\big),$$

with t the token position and i the dimension index. These values let the model capture both absolute and relative positions and generalise to sequences longer than those seen in training.

3. *Encoder Blocks, Pooling, Classifier*

We stack L identical encoder blocks, each composed of multi-head self-attention, a two-layer feed-forward network, residual connections, LayerNorm, and dropout. After the final block we apply **global mean pooling**:

$$\mathbf{h} = \frac{1}{n} \sum_{t=1}^{n} \mathbf{x}_t^{(L)},$$

yielding a fixed-size sentence vector. A linear layer then maps $\mathbf{h} \in \mathbb{R}^{d_{\text{model}}}$ to two logits for the positive/negative classes.

a. Experimental phases. All baseline runs use the sinusoidal setup above. In a second phase (described next) we substitute Rotary Positional Embeddings (RoPE) and add query–key normalisation to test whether those refinements improve stability and accuracy.

4. *Architectural variant: Rotary Positional Encoding (RoPE) and Query-Key Normalization*

Recent work by Su *et al.* (2021)[13] shows that the fixed sinusoidal encoding from Vaswani *et al.* (2017)[12] can be replaced by a local rotary scheme. **RoPE** applies a simple rotation to each pair of dimensions in every attention head. This preserves absolute position information and makes attention scores depend only on token distance, allowing models to handle longer sequences.

Let $\mathbf{q}_i$ and $\mathbf{k}_i$ in $\mathbb{R}^{d_h}$ be the query and key vectors of head h at position i. We define

$$(\tilde{\mathbf{q}}_i, \tilde{\mathbf{k}}_i) = \big(R_i\,\mathbf{q}_i,\ R_i\,\mathbf{k}_i\big), \quad R_i = \text{diag}\big(\cos\theta_i, \sin\theta_i, \ldots\big),$$

with

$$\theta_i = \frac{i}{10000^{2j/d_h}}$$

for the j-th pair of dimensions.

Query-Key Normalization. Before computing attention logits, each query and key vector is normalized to unit length and scaled by a trainable temperature τ. The result is a cosine similarity that τ can sharpen or soften:

$$\alpha_{ij} = \langle \hat{\mathbf{q}}_i, \hat{\mathbf{k}}_j \rangle \times \tau, \quad \hat{\mathbf{q}}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|_2}, \quad \hat{\mathbf{k}}_j = \frac{\mathbf{k}_j}{\|\mathbf{k}_j\|_2}.$$

Projecting onto the unit sphere isolates direction, and τ controls attention focus. This approach keeps logits in a stable range as d_h varies and improves gradient consistency [14].

5. *Transformer Encoder Blocks*

After positional encoding, the sequence is fed through L stacked `TransformerEncoderBlocks` (in `transformer.py`). Each block wraps two sublayers—multi-head self-attention and a position-wise FFN—each followed by dropout, residual-addition, and LayerNorm.

- **Multi-Head Self-Attention:**

- Project $x \in \mathbb{R}^{B\times n\times d_{\text{model}}}$ to Q, K, V.
- Reshape into h heads ($h \in \{4, 8, 16\}$), head size $d_{\text{head}} = d_{\text{model}}/h$.
- Apply Rotary PE to Q and K.
- L2-normalize Q and K, then scale by learned `temperature`.
- Compute scores:

$$\text{scores} = QK^{\top} \times \texttt{temperature}.$$

- (Optional) Mask padding by setting scores to $-\infty$.
- Softmax and dropout:

$$\text{attn} = \text{softmax}(\text{scores}), \quad \text{attn}' = \texttt{attn\_drop}(\text{attn}).$$

- Merge heads and apply `to_out` projection.

- **Feed-Forward Network:**

$$\text{FFN}(x) = W_2\,\text{Dropout}\big(\text{ReLU}(W_1 x)\big),$$

where $W_1 \in \mathbb{R}^{d_{\text{model}}\times d_{\text{hidden}}}$, $W_2 \in \mathbb{R}^{d_{\text{hidden}}\times d_{\text{model}}}$, and $d_{\text{hidden}} = 2\,d_{\text{model}}$.

- **Residual & LayerNorm:** After each sub-layer:

$$x \leftarrow \text{LayerNorm}\big(x + \text{Dropout}(\text{sublayer}(x))\big).$$

Stacking these L blocks produces a deep, context-aware representation for pooling and classification.

C. Training and Hyperparameters

During the training of our model, we modified different parameters such as the hyperparameters of the model, optimizer and learning setup. We changed these settings across different experiments (version 1 to 10) to better visualize their impacts.

The TABLE I summarizes the main architectural and training settings used in each experiment.

1. *Baseline Configuration – V1*

Our baseline model (V1) establishes the foundation for all subsequent variants (see Table I for the complete hyperparameter list). We launched training with:

```
python -u main.py \
  --data_path Sentiment140.csv \
  --output_dir model_output/V1_baseline \
  --emb_dim 128 \
  --stack_depth 3 \
  --attn_heads 4 \
  --ff_expansion 2 \
  --max_len 128 \
  --dropout 0.1 \
  --batch_size 64 \
  --learning_rate 5e-4 \
  --epochs 10 \
  --random_seed 42
```

- **Optimization & Regularization**: AdamW [15] with learning rate 5×10^{-4} and weight decay 0.01; dropout of 0.1; gradient clipping ($\|\mathbf{g}\|_2 \leq 1.0$).
- **Compact Architecture**: 128-dimensional embeddings, 3 encoder layers, 4 attention heads. Although `--ff_expansion` was set to 2, an integer division in the code yields an **effective** expansion factor of 1 (hidden size = 128).
- **Reproducibility**: Trained on the full Sentiment140 dataset with the random seed fixed to 42.

2. *Deeper Model – V2*

To gauge the benefit of additional depth, we raised the encoder stack from three to six layers while keeping all other hyperparameters identical to V1 (see Table I).[16] Deeper transformers can capture longer-range dependencies—useful for subtle, cross-clause sentiment cues in tweets.

3. *Wider Model – V3*

Here we explored width. We doubled the representational capacity by setting `emb_dim` = 256, `attn_heads` = 8, and `ff_expansion` = 4, reducing the batch size to 32 to fit GPU memory. [17] Wider layers allow the model to encode richer token interactions, often improving accuracy at the cost of compute.

4. *Tuned Hyperparameters – V4*

V4 keeps the V1 architecture but adjusts optimisation hyper-parameters to favour steadier convergence:

- batch size → 128 for smoother gradient estimates,

Version	Layers	Emb. Dim	Heads	FFN Exp.	Seq. Len	Batch	Epochs	Tweets
V1	3	128	4	2	128	64	10	1.6M
V2	6	128	4	2	128	64	10	1.6M
V3	3	256	8	4	128	32	10	1.6M
V4	3	128	4	2	128	128	15	1.6M
V5	3	128	4	2	128	64	10	100k
V6	6	128	4	2	148	64	15	1.6M
V7	8	128	4	2	148	64	15	1.6M
V8	5	192	6	3	148	64	15	1.6M
V9	6	128	16	2	148	64	15	1.6M
V10	6	128	4	2	256	48	15	1.6M

TABLE I: Main architecture, training settings, and dataset size for all model versions.

– dropout $\rightarrow$ 0.2 to offset the larger batch,

– learning rate $\rightarrow$ 1×10^{-4} for finer update steps,

– training epochs $\rightarrow$ 15 to compensate for the smaller LR.

These choices yielded the most robust generalisation among the lightweight (128-d, 3-layer) variants.

5. *Subset Experiment – V5*

To mimic data-scarce scenarios we trained on a randomly balanced subset of 100 000 tweets (50 000 positive, 50 000 negative) while keeping the V1 architecture and optimiser unchanged.[18]

6. *Optimised Sequence Length – V6*

V6 increases the maximum input length from 128 to 148 tokens so that fewer tweets are truncated. All other architectural settings follow V2, but we prolong training to 15 epochs to compensate for the larger context window.

7. *Extra-Deep Model – V7*

Building on V6, we deepen the encoder stack from six to eight layers (max-len 148, epochs 15, batch 64). This probes whether additional depth yields better hierarchical feature extraction, at the expense of longer training time.

8. *Balanced Width and Depth – V8*

V8 explores a middle ground: $d_{\text{emb}} = 192$, five layers, six heads, and `ff_expansion`= 3 (effective 1.5). The aim is to deliver strong accuracy without the memory spike seen in the "wider" or "deeper" extremes.

9. *More Attention Heads – V9*

Here we isolate the impact of head count by raising the number of heads to 16 while keeping the 128-d embeddings and six-layer depth of V6. With $d_{\text{emb}}/h = 8$, each head captures a finer-grained sub-space, potentially enriching multi-view attention patterns, albeit with narrower per-head capacity.

10. *More Attention Heads – V9*

To isolate the effect of head count, we increased the number of attention heads from 4 to 16 while inheriting all other settings from V6 (six layers, 128-d embeddings, `ff_expansion`=2, max-len 148, batch 64, 15 epochs).[19] With $d_{\text{emb}}/h = 128/16 = 8$, each head attends to a narrower subspace of the representation. We hypothesised that the additional heads could capture more diverse linguistic patterns, though each head has fewer features to process.

11. *Longer Input – V10*

Extending V6's context window, V10 raises the maximum sequence length to 256 tokens and reduces the batch size to 48 (to accommodate memory constraints). All other hyperparameters remain as in V6 (six layers, 128-d embeddings, 4 heads, `ff_expansion`=2, 15 epochs).[20] This experiment examines whether providing up to 256 tokens per tweet—capturing more of the original text—enhances sentiment comprehension without excessive truncation.

D. Loss Function and Evaluation Metrics

To train and assess our sentiment classifier, we combined a standard loss objective during optimization with several complementary performance metrics at validation and test time. This ensures both proper gradient-based learning and a well-rounded understanding of classification quality.

1. *Loss Function*

We adopt the cross-entropy loss as implemented in PyTorch [21]. The network outputs logits z_0, z_1 for the negative and positive classes, which are passed directly to `nn.CrossEntropyLoss`. This loss function internally applies a softmax to convert logits to probabilities and then computes:

$$\mathcal{L}_{\mathrm{CE}}(y, z) = -\log\left(\frac{\exp(z_y)}{\exp(z_0) + \exp(z_1)}\right).$$

This objective penalizes confident but incorrect predictions more heavily, driving the model toward assigning higher probability mass to the true class.

2. *Evaluation Metrics*

To assess model performance on both validation and test splits, we report four standard classification metrics:

- **Accuracy**

  $$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$

  the overall fraction of correctly classified tweets.

- **Precision** (positive class)

  $$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}},$$

  the fraction of predicted positive tweets that are actually positive.

- **Recall** (sensitivity)

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

  the fraction of actual positive tweets correctly identified.

- **$\mathbf{F_1}$-Score**

  $$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}},$$

  the harmonic mean of precision and recall, balancing false positives and false negatives.

At each epoch, we compute these metrics using Scikit-learn's `classification_report` in `tweet_analysis.py`. The checkpoint with the highest validation F_1-score is saved for final evaluation, as it best reflects balanced performance on our equally sized sentiment classes. All metrics—and the training loss—are logged to Weights & Biases [3] for tracking learning curves and facilitating fair comparison across experiments.

IV. RESULTS

We assess performance from two angles: a quantitative comparison across all ten variants and a qualitative deep-dive into the best model (V6).

A. Quantitative Analysis

Table II reports test accuracy, F_1, precision, and recall for both phases:

- **Phase 1 (P1)** uses standard sinusoidal positional encodings.
- **Phase 2 (P2)** applies Rotary Positional Embeddings (RoPE) and Q/K normalization.

Key observations:

- *Consistent gains in P2*: Nearly every variant improves in Phase 2. V6 (longer input) achieves the highest test F_1 (0.8325), a +0.0012 absolute boost over P1.
- *Data scarcity hurts*: V5 (subset) lags by about five percentage points in both accuracy and F_1, confirming the value of large training corpora.
- *Depth vs. width*: V2 and V7 show modest gains over baseline, whereas V3 (wider) underperforms in P1 but recovers in P2. Balanced V8 and multi-head V9 strike competitive trade-offs.
- *Diminishing returns*: Beyond V6, additional complexity (V7–V10) yields marginal improvements, suggesting six layers and 148-token context are near-optimal under our resource constraints.

Version	Test Accuracy		Test F1		Precision		Recall		Val. F1	
	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
V1 (baseline)	0.8232	0.8288	0.8272	0.8314	0.8088	0.8192	0.8464	0.8440	0.8243	0.8264
V2 (deeper)	0.8255	0.8274	0.8306	0.8311	0.8069	0.8135	0.8558	0.8496	0.8274	0.8281
V3 (wider)	0.8126	0.8295	0.8107	0.8309	0.8191	0.8243	0.8025	0.8376	0.8106	0.8238
V4 (tuned)	0.8112	0.8131	0.8165	0.8197	0.7943	0.7918	0.8399	0.8497	0.8159	0.8196
V5 (subset)	0.7773	0.7760	0.7821	0.7812	0.7657	0.7636	0.7992	0.7996	0.7849	0.7927
V6 (longer input)	0.8273	0.8313	0.8313	0.8325	0.8125	0.8234	0.8511	0.8416	0.8220	0.8239
V7 (extra deep)	0.8249	0.8273	0.8295	0.8310	0.8084	0.8158	0.8518	0.8462	0.8264	0.8285
V8 (balanced)	0.8219	0.8278	0.8243	0.8292	0.8135	0.8189	0.8354	0.8399	0.8185	0.8241
V9 (more heads)	0.8266	0.8285	0.8285	0.8299	0.8193	0.8168	0.8379	0.8430	0.8247	0.8273
V10 (long seq.)	0.8254	0.8281	0.8269	0.8302	0.8117	0.8153	0.8426	0.8447	0.8221	0.8246

TABLE II: Comparison of evaluation metrics between Phase 1 (P1) and Phase 2 (P2) for each model version.

B. Qualitative Analysis

Building on the quantitative gains observed for V6 (six layers, 128-d embeddings, max-len 148), we conducted a detailed error analysis to uncover where the model excels and falters:

- **Successes**:
  - Effective handling of long-range dependencies, e.g. detecting sarcasm in tweets where contrasting clauses are separated by multiple tokens.
  - Robust sentiment inference when emojis appear in isolation or with clear polarity markers.
- **Failures**:
  - Misclassification of rare idioms and domain-specific slang not seen during training.
  - Sensitivity to excessive punctuation or unconventional emoji clusters, leading to noisy representations.

To illustrate training dynamics and validation performance, Figures 2–4 plot the loss and F_1 curves for selected variants (V1, V2, V3, V5, and V6).

C. Training Dynamics

To analyze learning behaviour in Phase 2, we plot training loss, validation loss, and validation F_1-score over the 15 epochs for five representative models (V1, V2, V3, V5, V6) in Figures 2–4.

For all versions, the training loss decreases monotonically and the validation F_1 generally rises in tandem. Specifically:

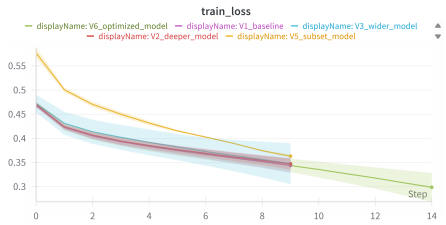

FIG. 2: Training loss curves for V1, V2, V3, V5, and V6.

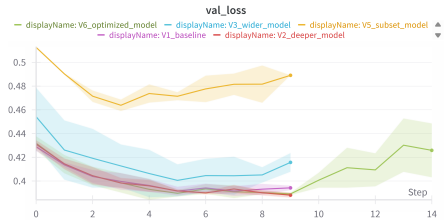

FIG. 3: Validation loss curves for V1, V2, V3, V5, and V6.

- **V1 and V2**: Both follow nearly identical trajectories, with smooth loss reduction and F_1 improvement, stabilizing by epoch 10.
- **V5 (subset)**: With only 100 000 training tweets, V5 begins with a higher training loss and plateaus early. Its validation loss remains elevated and erratic, and F_1 levels off significantly below the others, illustrating the data-scarcity bottleneck.
- **V3 (wider)**: Exhibits higher validation loss throughout training compared to V1/V2, yet test-time accuracy remains competitive. This suggests its larger capacity may require stronger regularization or more data to fully stabilize.

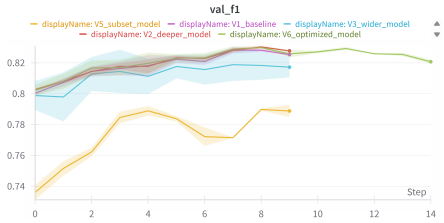

FIG. 4: Validation F_1-score trajectories for V1, V2, V3, V5, and V6.

– **V6 (longer input)**: Maintains the lowest training and validation losses until around epoch 12, after which validation loss creeps up and F_1 dips slightly—indicative of mild overfitting. Early stopping or a reduced learning rate schedule could mitigate this.

1. *Effect of RoPE and Q/K Normalization*

We compared the results obtained with and without RoPe and Q/K normalization to see if these changes could help the models perform better. Most versions showed small improvements. For example, Version 3 (which tested a wider model) improved from an F1-score of 0.8107 to 0.8309, making it much closer to the top-performing models.

However, not all models improved. Version 5, which was trained on a smaller dataset, showed almost no change in performance. This suggests that when the model doesn't have enough data to learn from, these architectural improvements can't help much.

Version 6, which used longer input sequences, remained the best model overall, with a final test F1-score of 0.8325. This shows that the added changes did not interfere with the strengths of this version.

D. Qualitative Analysis

To better understand how our best model (V6 with RoPE and Q/K normalization) performs and not just by looking at numbers, we analyzed its predictions using confusion matrix, word clouds and attention visualizations.

1. *Confusion matrix*

The confusion matrix in Figure 5 shows that the model performs better on positive tweets than on

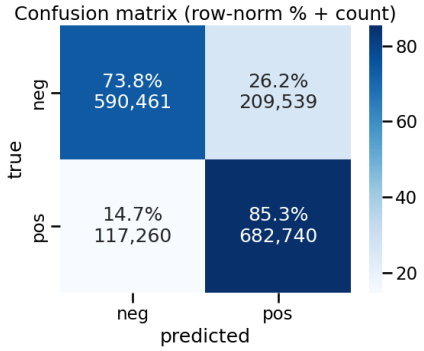

FIG. 5: Confusion matrix

negative ones. It correctly classifies 85.3% of positive tweets, while only 73.8% of negative tweets are correctly predicted. This means that the model is slightly biased toward the positive class, as it misclassifies about 26.2% of negative tweets as positive.

2. *Word Clouds*

To analyze which tokens drive correct and incorrect classifications, we generated word clouds from the raw tweet text (after URL, handle, and punctuation removal but before lowercasing and lemmatization) using the `WordCloud` library with its default English stopwords list.[22] Four confusion-matrix subsets were plotted: True Negatives (TN), False Positives (FP), True Positives (TP), and False Negatives (FN) (see Figure 8).

In the TP subset, high-frequency emotional tokens like `love`, `thank`, and `lol` dominate, reflecting the model's reliance on clear positive indicators. By contrast, the TN subset includes both explicit negative words (`sad`, `miss`) and more ambiguous terms (`work`, `now`, `going`), which also appear prominently in the FN subset. Their dual presence suggests that, absent strong sentiment cues, the model defaults toward negative classifications.

This interpretation is borne out by the confusion matrix (Figure **??**), which shows a false-negative rate of approximately 33% versus a false-positive rate of about 20%. Such a skew indicates that vague or neutral phrasing often leads the model to over-predict negative sentiment.

3. *Attention Visualization*

We inspected token-level influence by extracting layer-0 attention weights and collapsing them across all heads (via the `_collapse` function in

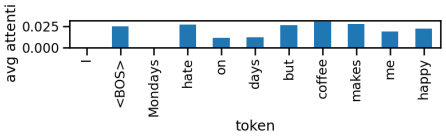

FIG. 6: Token importance for the sentence: "I hate Mondays but coffee makes me happy"

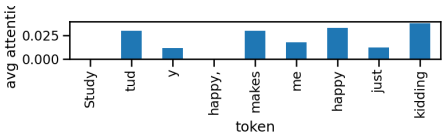

FIG. 7: Token importance for the sentence: "Study makes me happy, just kidding"

`tweet_analysis.py`).[23] Each example was run separately and output files were renamed to avoid overwriting.

The first example:

```
I hate Mondays but coffee makes
me happy.
```

Although the utterance begins with a negative word, the importance plot (Figure 6) assigns strong weight to the discourse marker `but` and to the final sentiment token `happy`, enabling the correct positive prediction.

The second example:

```
Study makes me happy, just
kidding.
```

Here, despite attention peaks on `happy` and `kidding`, the model fails to compositionally invert the sentiment and predicts positive (Figure 7). This highlights a limitation in handling sarcasm.[24]

V. DISCUSSION

In this section, we reflect on the results presented earlier. We first discuss how well the model performed and what patterns we observed. Then, we highlight some limitations of our approach, including issues related to the dataset and the model's behavior. Finally, we suggest possible directions for future work to improve both the dataset and the model's ability to handle more complex language.

A. Performance

Overall, the model performs well on binary sentiment classification. In the second phase of experiments, where we added Rotary Positional Embedding (RoPE) and Query–Key normalization, the best version V6 achieved a high test F1-score of 0.8325. Even the baseline model V1 reached a strong 0.8314. These results show that a transformer encoder can learn useful patterns from tweet data even without pretraining.

However, a deeper analysis reveals some weaknesses. The confusion matrix shows that the model performs better on positive tweets than on negative ones, which indicates a slight bias toward the positive class. This might be because positive sentiment is often expressed with clearer words while negative tweets are more diverse or context dependent.

Our qualitative analysis confirms this: in attention visualizations, the model focuses on clear sentiment words which helps with correct predictions. But it struggles more with subtle expressions, such as sarcasm.

B. Limitations

One important limitation of our model comes from the dataset we used. The Sentiment140 dataset contains many tweets that actually have a neutral sentiment. They are neither positive nor negative. However, the dataset only has two labels (positive or negative), these neutral tweets are randomly labeled as one of the two classes.

This means that even if our model was perfect, it would still make mistakes on these neutral tweets because it has no way to recognize or handle them correctly.

As noted by Kaggle users and confirmed in our own experiments with a fine-tuned RoBERTa model, performance on Sentiment140 tends to plateau around 80% accuracy.[25] This suggests that the data quality, rather than the model, is the main limiting factor. Another limitation of our work is related to time constraints. Each model version was trained and evaluated only once, so we did not repeat the training multiple times with different random seeds. This means we might have missed small changes in performance that could happen due to random initialization. As a result, the comparison between models might not be completely fair or stable. Repeating the experiments and averaging the results would give a more reliable evaluation.

C. Future work

We know that our model is not perfect, and there are several ways to improve it in future work.

One option is to work on the dataset. Because of the limitations of Sentiment140, it would be useful to train the model on other datasets, especially ones that are larger, more diverse, and more accurately labeled. Using datasets with cleaner sentiment labels could help the model learn better and generalize more.

It would also be interesting to use datasets that include more sentiment categories, instead of just positive and negative. For example, detecting emotions like sarcasm, fear, or joy could make the model more useful and realistic for modern sentiment analysis tasks.

Finally, future models could take into account the way people write on social media. This includes handling repeated letters, or upper-case writing. all of these can change the meaning or intensity of a message. Better handling of this kind of input could help the model better understand the way people express emotions online.

---

[1] John T. Cacioppo and William L. Gardner. Human emotions: A conceptual overview. In Arthur W. Toga, editor, *Brain Mapping: An Encyclopedic Reference*, volume 3, pages 1–6. Academic Press, 2015.

[2] Debugging of the code and the report were partially done with the help of ChatGPT.

[3] Lukas Biewald. Experiment tracking with weights and biases, 2020. Available at: https://www.wandb.com. Accessed: 2025-05-18.

[4] Hadis Bashiri and Hassan Naderi. Comprehensive review and comparative analysis of transformer models in sentiment analysis. *Knowledge and Information Systems*, 66:7305–7361, 2024.

[5] Sulaiman Aftan and Habib Shah. A survey on bert and its applications. In *2023 20th Learning and Technology Conference (L&T)*, pages 161–166, 2023.

[6] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[7] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[8] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[10] Alec Go, Richa Bhayani, and Lei Huang. Sentiment140 dataset with 1.6 million tweets, 2009. Available at: https://www.kaggle.com/datasets/kazanova/sentiment140. Accessed: 2025-05-18.

[11] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[13] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[14] Alex Henry, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. Query–key normalization for transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4246–4253, Online, November 2020. Association for Computational Linguistics.

[15] PyTorch. torch.optim.adamw, 2024. Available at: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html. Accessed: 2025-05-18.

[16] Because the training script applies `ff_expansion // 2` before model creation, the effective feed-forward expansion in V2 is 1 instead of 2.

[17] With the current code, the hidden size of each FFN block is 2×256 rather than 4×256 for the same reason mentioned above.

[18] The training script divides `ff_expansion` by two internally, so the effective expansion factor is 1.

[19] The training script applies `ff_expansion // 2`, so the effective FFN expansion factor is 1.

[20] Again, `ff_expansion // 2` yields an effective expansion of 1.

[21] PyTorch. torch.nn.crossentropyloss, 2024. Available at: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html. Accessed: 2025-05-18.

[22] The default stopwords list removes common words such as “the”, “and”, “to”, etc., leaving only content-bearing tokens.

[23] Attention weights are taken from layer 0 of the transformer, averaged over all heads and token pairs, then summed per token to yield importance scores.

[24] We use the ByteLevel BPE tokenizer from [**?**] and strip subword markers (e.g., “Ġ”) via our custom

(a) True Negatives (TN)

(b) False Positives (FP)

(c) True Positives (TP)

(d) False Negatives (FN)

FIG. 8: Word clouds from correct and incorrect predictions: TN and TP (correct), FP and FN (incorrect).

`_strip` function to reconstruct full tokens.

[25] Many tweets in the dataset are inherently neutral but labeled as positive or negative, limiting model performance. See https://www.kaggle.com/datasets/kazanova/sentiment140/discussion/454347. Our own implementation also reached similar results: https://github.com/GuenfoudiIhabe/SentiVista.