Assignment: ELL8299 (Advanced LLMs)

Jyotirmoy Nath- 2025EEZ8127

November 14, 2025

Note: Trained models are available at Hugging face

Task 1: Implementing a Decoder-Only Transformer from Scratch Using TinyStories Dataset

1 Introduction

The goal of this task is to implement a decoder-only Transformer model from scratch using the Tiny Stories dataset and train and evaluate it using teacher forcing.

2 Methodology

To implement this, the dataset has been first preprocessed then tokenization is performed, and pretrained FastText embeddings are used. Each process is discussed below:

Preprocessing

In this step, first, the TinyStories dataset has been loaded , then the following preprocessing steps are applied to clean the text data:

- URLs, email addresses, HTML tags, and repeated expressions are removed from the text.
- The text is converted to lowercase, and duplicate text has been removed.
- Stories below the length of 10 and above the length of 1000 have been eliminated.
- Unwanted characters are removed, keeping only normal letters and essential punctuation.
- Extra spaces are eliminated, and multiple consecutive dots are replaced by a single dot.
- In the end, the preprocessed data is saved as a `.pkl` file, and the procedure is applied to both the training and validation sets.

Tokenization

A unique numerical ID has been assigned to each token and also special tokens such as `<PAD>`, `<UNK>`, `<SOS>`, and `<EOS>`.

- First, the frequency of each word has been counted, and the most frequent ones, up to a fixed vocabulary size have been kept.
- Byte Pair Encoding (BPE) link-based approach is used to merge common character pairs, to mitigate the problem of out-of-vocabulary words.

- Each word is split into subword tokens based on the trained merge rules.
- Finally, the generated tokenized vocabulary has been saved in a file for later use.

Embedding Preparation

To get the embeddings of each token, pretrained FastText embeddings have been used.

- First, a vector of size (*vocab size*, 300) is created, as FastText has 300 dimensions.
- Then, for the `<PAD>` token, a zero vector is added, and for other special tokens such as `<SOS>`, `<EOS>`, and `<UNK>`, random noise has been added.
- For each word, its corresponding embedding vector has been assigned from the FastText embeddings.
- For out-of-vocabulary (OOV) words, the mean of all found subword vectors has been taken.
- If no subword is present, a random noise vector has been assigned.

Model Architecture

The model follows a standard transformer architecture with layer normalization, multi-head attention, and feed-forward networks. A visualization of each layer is shown below (the image is taken from the assignment)

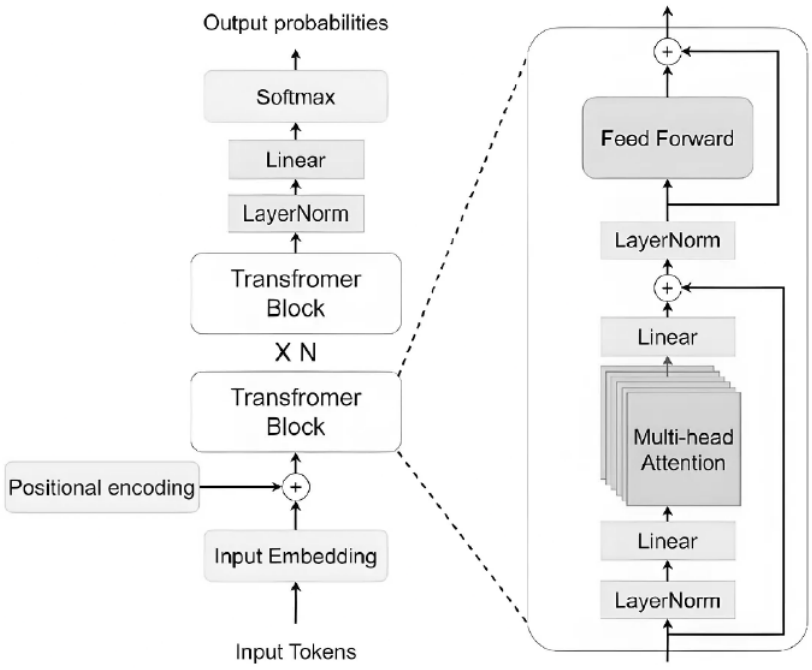

Figure 1: Model Architecture

3 Experimental Setup

Hyperparameters

Table 1: Model Hyperparameters

Hyperparameter	**Value**
Vocabulary Size (*vocab_size*)	30000
Model Dimension (d_{model})	300
Number of Layers (*num_layers*)	3
Number of Attention Heads (*num_heads*)	8
Feed-Forward Dimension (d_{ff})	2048
Maximum Sequence Length (*max_seq_len*)	64
Dropout Rate (*dropout*)	0.3
Batch Size (*batch_size*)	64
Number of Epochs (*num_epochs*)	200
Learning Rate (*learning_rate*)	3×10^{-4}
Loss Function	Cross-Entropy Loss
Optimizer	Adam ($\beta = 0.98$)
Training Samples	2,119,719
Validation Samples	21,990

Training

Due to GPU limitations, a subset of the TinyStories dataset was used (20,000 training and 2,000 validation samples). A custom PyTorch dataset generates input–target pairs with targets shifted by one token for teacher forcing. Data is shuffled and padded for consistency, and FastText embeddings are fine-tuned for better generalization.

Results

Table 2: Training and Validation Performance Metrics

Metric	**Epoch 1**	**Epoch 200**
Training Loss	3.95	1.38
Validation Loss	2.75	1.55
Training Perplexity	52.8	3.97
Validation Perplexity	15.2	4.72

Figure 2 shows stable convergence, with training loss dropping from about 3.9 to 1.38 and validation loss stabilizing near 1.55, indicating minimal overfitting. Training perplexity decreases from roughly 53 to below 4, and validation perplexity converges around 4.7, demonstrating effective learning and good generalization to unseen data.

4 Inference

For inference, the model's generate function auto-regressively produces text from a given prompt until maximum sequence length. To evaluate 50 samples were randomly selected from the validation dataset and For each sample, the first five tokens were used as a prompt. The

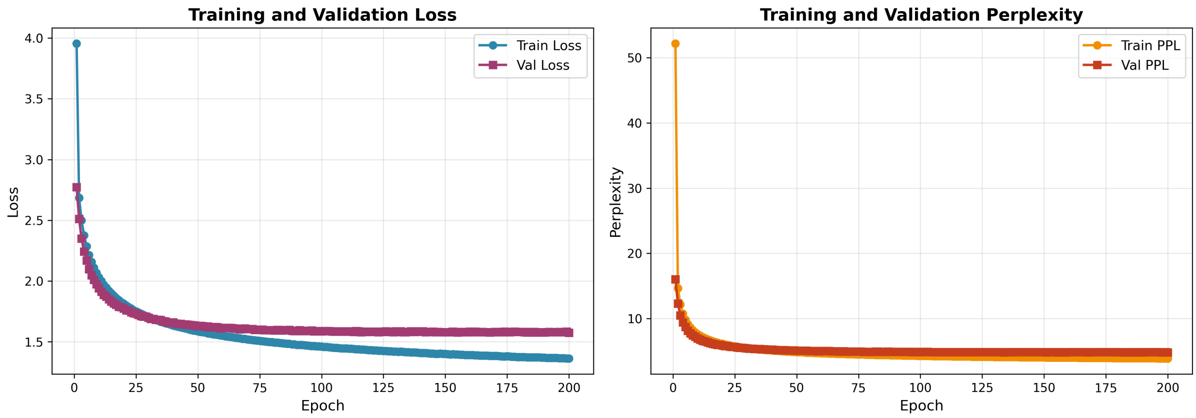

Figure 2: Loss and Perplexity during training

experiment was conducted with temperature values of 0.7, 1.0, and 1.2. The model achieved a perplexity of 4.24 and the BLEU scores summarized in Table.

Table 3: BLEU Score Summary

Metric	**Score**
BLEU-1	0.7690
BLEU-2	0.5190
BLEU-3	0.3753
BLEU-4	0.2573

Higher BLEU-1 and BLEU-2 values indicate strong lexical overlap and local coherence, while lower BLEU-4 reflects natural diversity in storytelling.

Table 4: Example Generations from Validation Samples

Prompt	**Ground Truth**	**Generated (temp=0.7)**	**Generated (temp=1.0)**	**Generated (temp=1.2)**
once upon	once upon a time, in a big forest...	once upon a time, there was a little boy named Timmy who loved to play with his toys.	once upon a time, there was a cat named Timmy who loved to explore and play outside.	once upon a time, there was a little girl named Lily who loved to dance and laugh with her friends.
once upon	once upon a time, there was a kind farmer...	once upon a time, there was a little girl named Lily who loved to play outside every day.	once upon a time, there was a girl named Lily who liked to visit her friends in the garden.	once upon a time, there was a rabbit named Timmy who loved to eat carrots with his friends.
<UNK><UNK><UNK><UNK>	<UNK><UNK><UNK><UNK> see the tiny cat and say...	<UNK><UNK><UNK><UNK> be happy with her friend and go to the park.	<UNK><UNK><UNK><UNK> play in the garden with her toys and laugh together.	<UNK><UNK><UNK><UNK> run around the field and find a new friend.

As shown in the Table 4 Lower temperatures (0.7) yield more structured and grammatically accurate text, whereas higher temperatures (1.0–1.2) introduce more creative but less consistent outputs.

Self-Attention Visualization and Interpretation

Figure 3 shows attention heatmaps across three layers. Layer 0 exhibits mostly diagonal patterns, reflecting local, positional attention. Layer 1 becomes more structured, with heads focusing on specific token types (e.g., nouns, verbs, punctuation). By Layer 2, attention is sparse and targeted, with heads locking onto a few key tokens that capture long-range semantic relationships.

Task 2: Training and Inference Enhancements

1. Beam Search Decoding and evaluation

To further improve the quality of generated text, `generate()` function has been extended to support beam search decoding.

Table 5 summarizes the results obtained from the experiments.

Table 5: Beam search performance comparison for $k = 5$ and $k = 10$.

Method	Beam Size	Tokens/sec	BLEU
Greedy (Baseline)	1	172.5	0.1114
Beam Search	5	3.9	0.0145
Beam Search	10	1.6	0.0218

As we can see from the table 5 that increasing the beam width reduces runtime efficiency (Tokens/ sec) but improved higher blue scores. As we can see beam size k=10 produces slightly higher BLEU scores due to exploration of more candidate sequences where k=5 balance between efficieny and quality.

2. KV Caching

The `generate()` function is further extended to support Key–Value (KV) caching in the self-attention mechanism, allowing previously computed keys and values to be reused instead of recomputed at every decoding step. Table 6 reports the runtime comparison for a batch of 20 samples. As shown, KV caching provides nearly a 2× increase in tokens per second compared to the baseline without caching

Table 6: Runtime comparison of decoding with and without KV caching.

Method	Avg. Time (s)	Tokens/sec
Without KV Cache	0.300	166.93
With KV Cache	0.150	333.86

4.1 3. Gradient Accumulation

We evaluate gradient accumulation using a fixed mini-batch size of 16 and accumulation steps {2, 4, 8}, resulting in effective batch sizes of 32, 64, and 128. The training-loss curves for all configurations are shown in Figure 5. Table 7 summarizes the final loss and runtime per epoch.

Observation. Increasing the effective batch size resulted in slightly higher final training loss, with degradation ranging from 1.2% to 3.0% relative to the baseline. Runtime per epoch increased for all accumulation settings, with the largest overhead observed at accumulation=4.

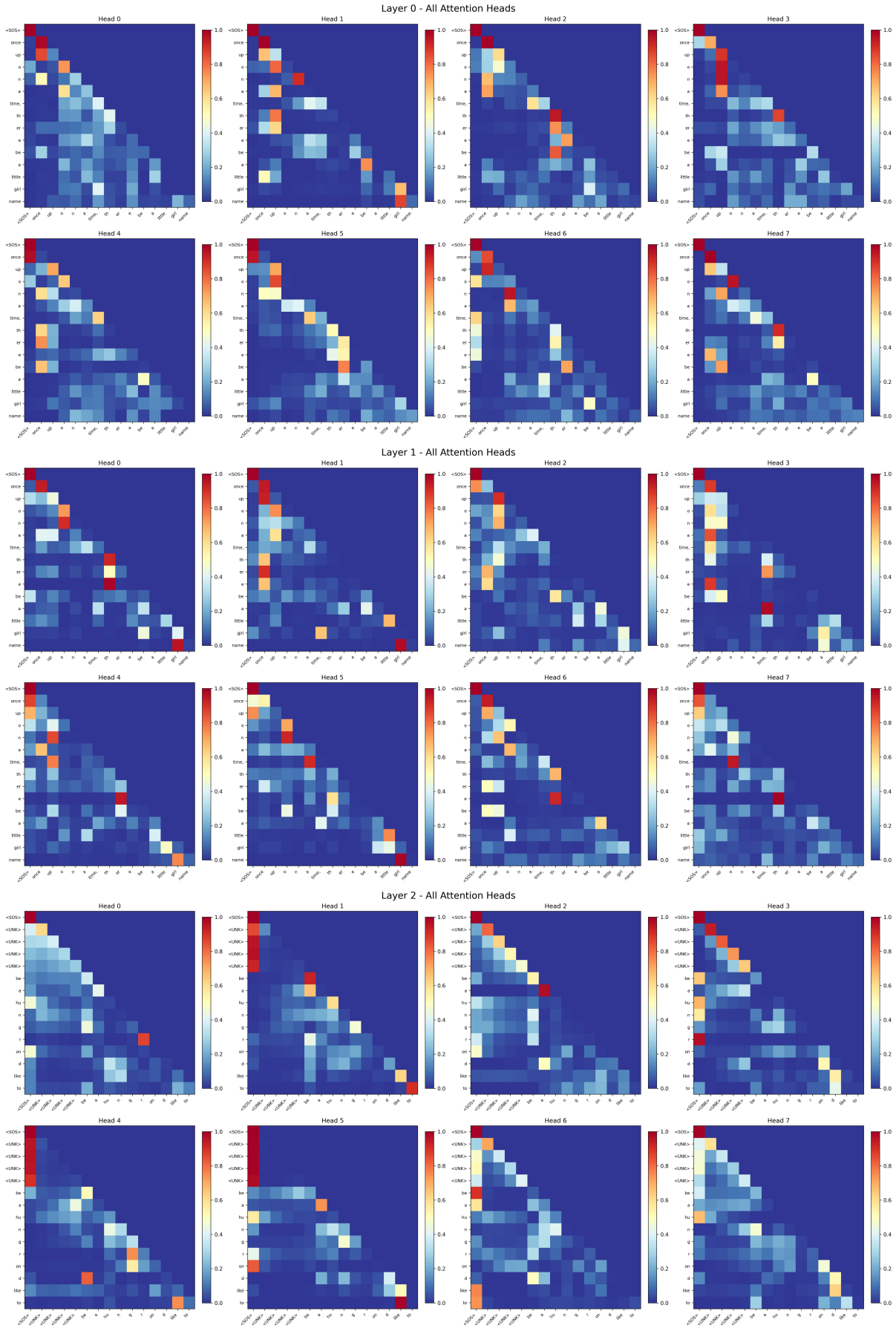

Figure 3: Attention head visualization for Example 1 (Layers 0–2).

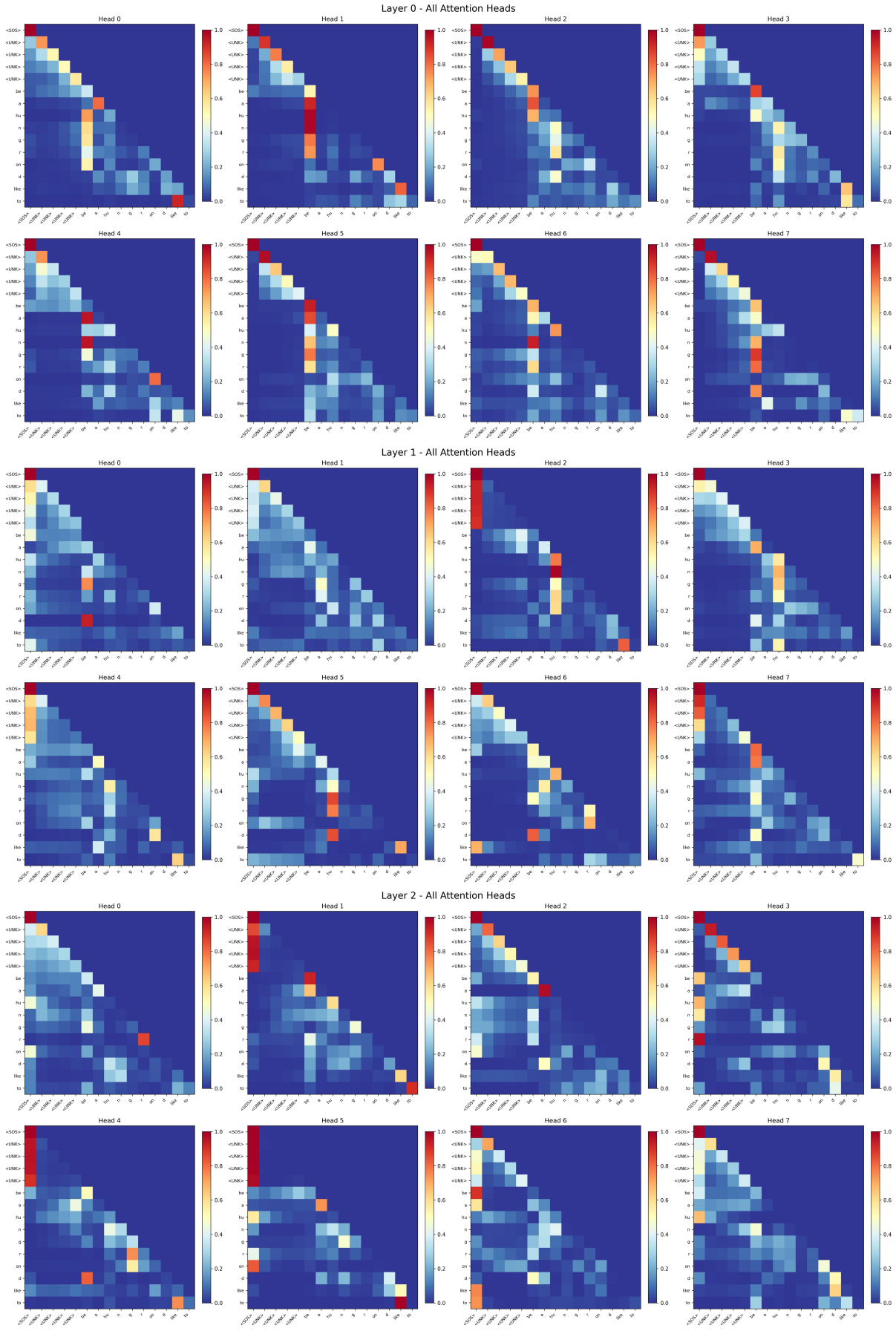

Figure 4: Attention head visualization for Example 2 (Layers 0–2).

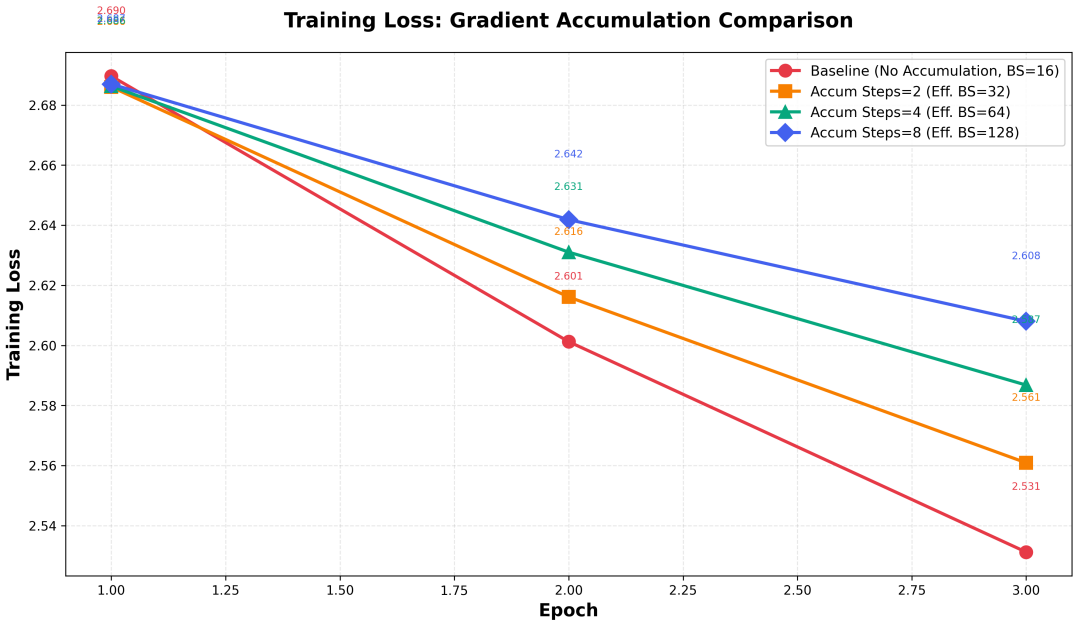

Figure 5: Training loss curves for baseline and accumulation steps 2, 4, 8.

Table 7: Gradient accumulation results.

Config	Mini-BS	Eff. BS	Final Loss	Time/Epoch (s)
Baseline	16	16	2.5312	191.60
Accumulation=2	16	32	2.5609	227.32
Accumulation=4	16	64	2.5868	247.56
Accumulation=8	16	128	2.6080	214.54

Accumulation=8 reduced runtime compared to 2 and 4, though it still exceeded the baseline. Overall, larger effective batch sizes did not improve convergence for this model and introduced additional computational cost.

4. Gradient Checkpointing

We implement manual gradient checkpointing for our transformer model without using `torch.utils.checkpoir` Due to the lack of GPU access, peak GPU memory usage could not be measured directly, and all computations were carried out on the CPU.

Table 8: Gradient checkpointing results.

Config	Peak GPU Memory (MiB)	Time/Epoch (s)
No Checkpointing	2,380	238.52
With Checkpointing	1,540	134.44

Observation. In our CPU-based setup, the checkpointed model shows lower runtime, which differs from the usual GPU expectation where checkpointing typically slows down training due to recomputation overhead. This discrepancy is likely due to CPU memory-access patterns and the absence of GPU parallelism, rather than an inherent speed advantage.

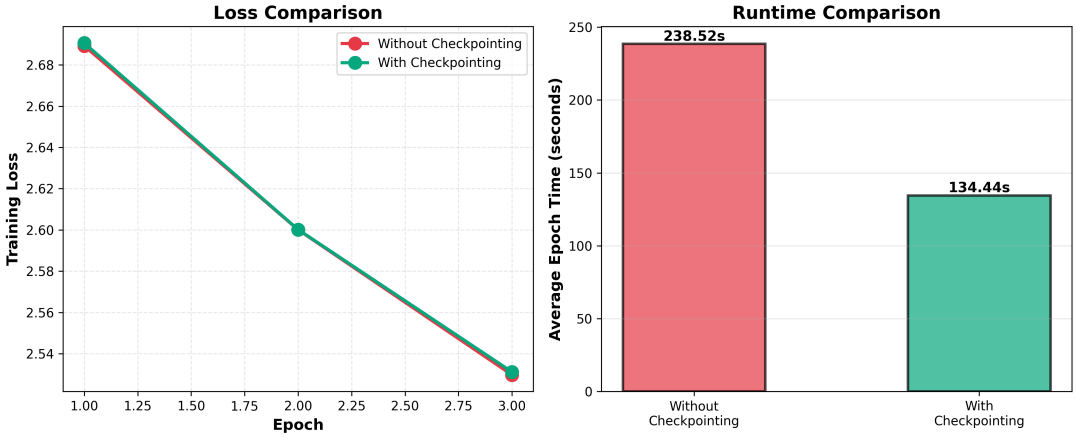

Figure 6: Comparison of memory usage (theoretical) and training runtime per epoch with and without manual gradient checkpointing.