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)

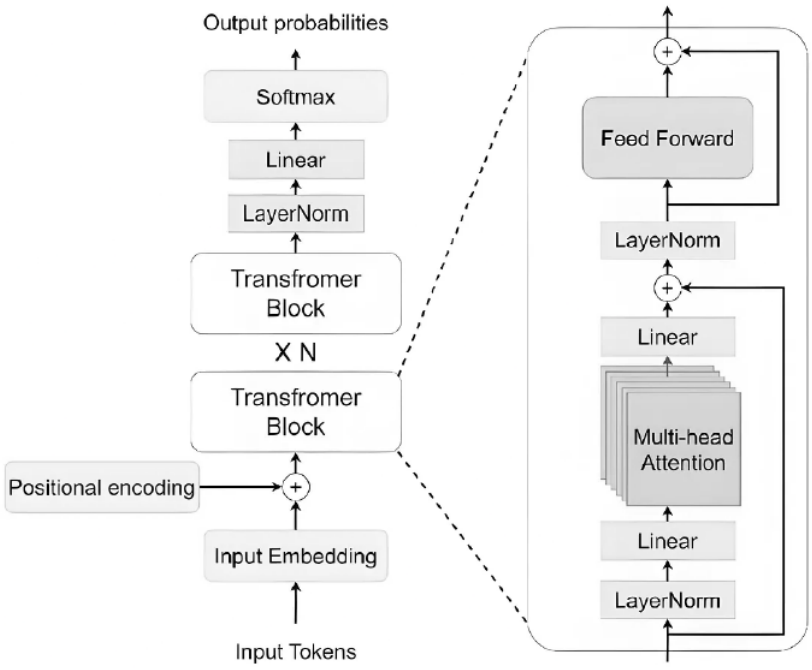

Figure 1: Model Architecture

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, we used a subset of TinyStories (20K train, 2K val). A custom PyTorch dataset creates shifted input–target pairs for teacher forcing, with shuffling and padding. FastText embeddings are fine-tuned. Two models are trained and uploaded to Hugging Face: one with 200 epochs, and a lighter version with 10 epochs for faster inference.

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

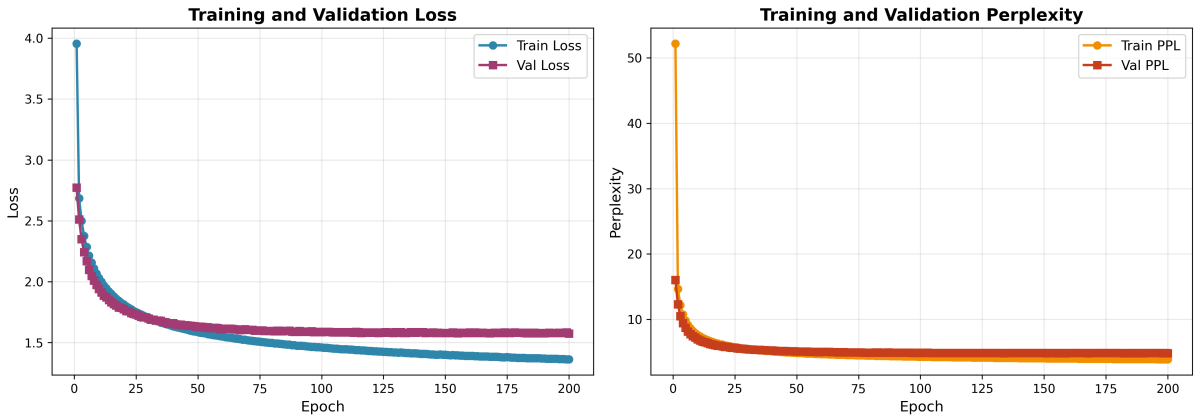

Figure 2: Loss and Perplexity during training

experiment was conducted with temperature values of 0.7, 1.0, and 1.2. The model achieved a **perplexity of 4.24** and **the average 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)
<UNK><UNK>	<UNK><UNK><UNK><UNK> see the tiny c...	<UNK><UNK><UNK><UNK>be very <UNK>ay with his fam in the park and explore it have a big...	<UNK><UNK><UNK><UNK>it be a girl who love to build thing with her good and with...	<UNK><UNK><UNK><UNK>be in the forevery imay she and many with their mum have...
once upon	once upon a time, in a big fore...	once upon a time, there be a little boy name lily. she love to eat yummy and play with his friend...	once upon a time, there be a little girl name lily. she love to fly out the sun for her...	once upon a time, there see two friend, once up...
once upon	once upon a time, in a small yard, there be a daisy...	once upon a time, there be a little boy name timmy. timmy love to play with his friend...	once upon a time, there be a little girl name sue. lily have a mother and her ter...	once upon a time, there be a little boy name ship th...

As shown in 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 fo-

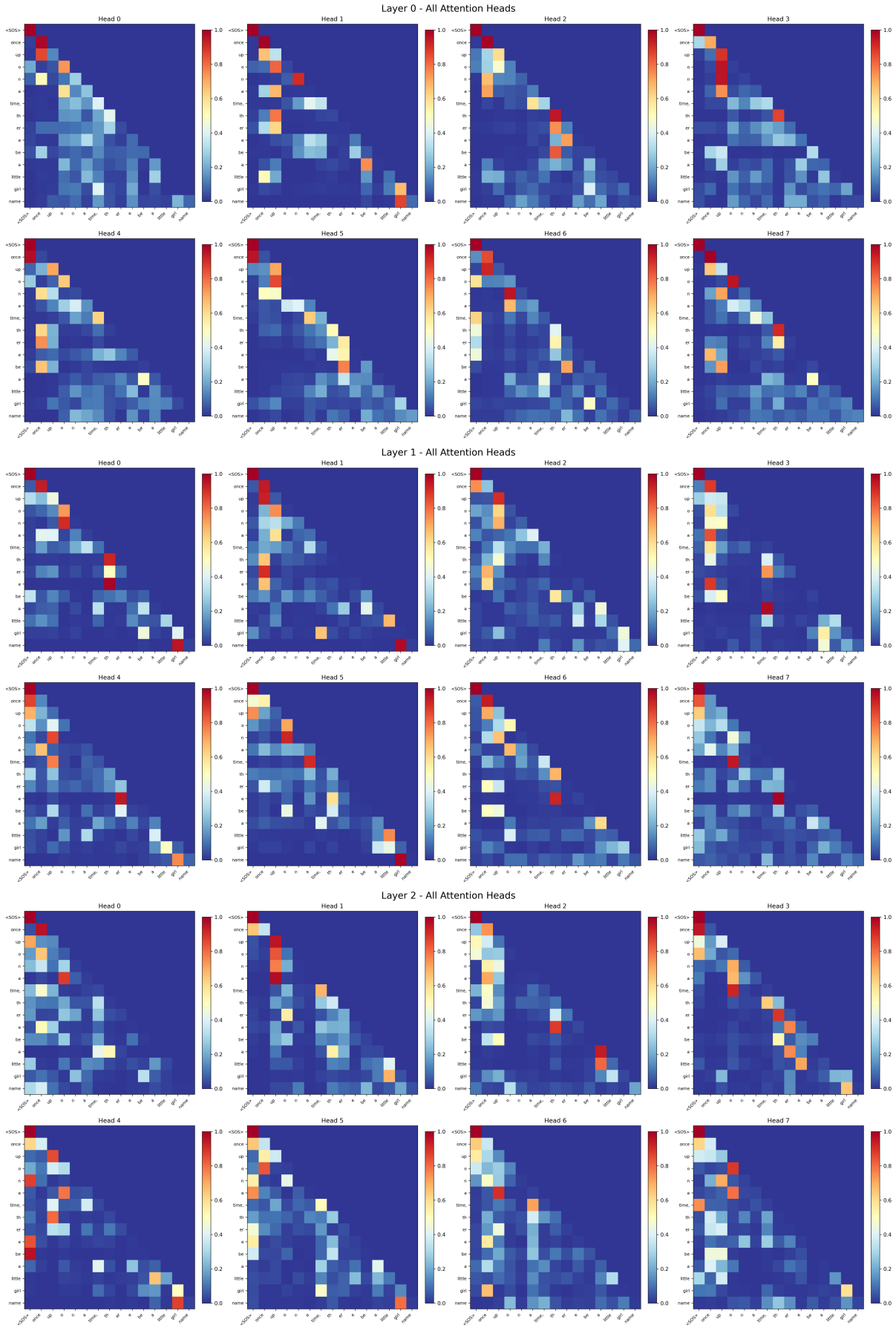

Figure 3: Attention head visualization for Example 1 (Layers 0–2).

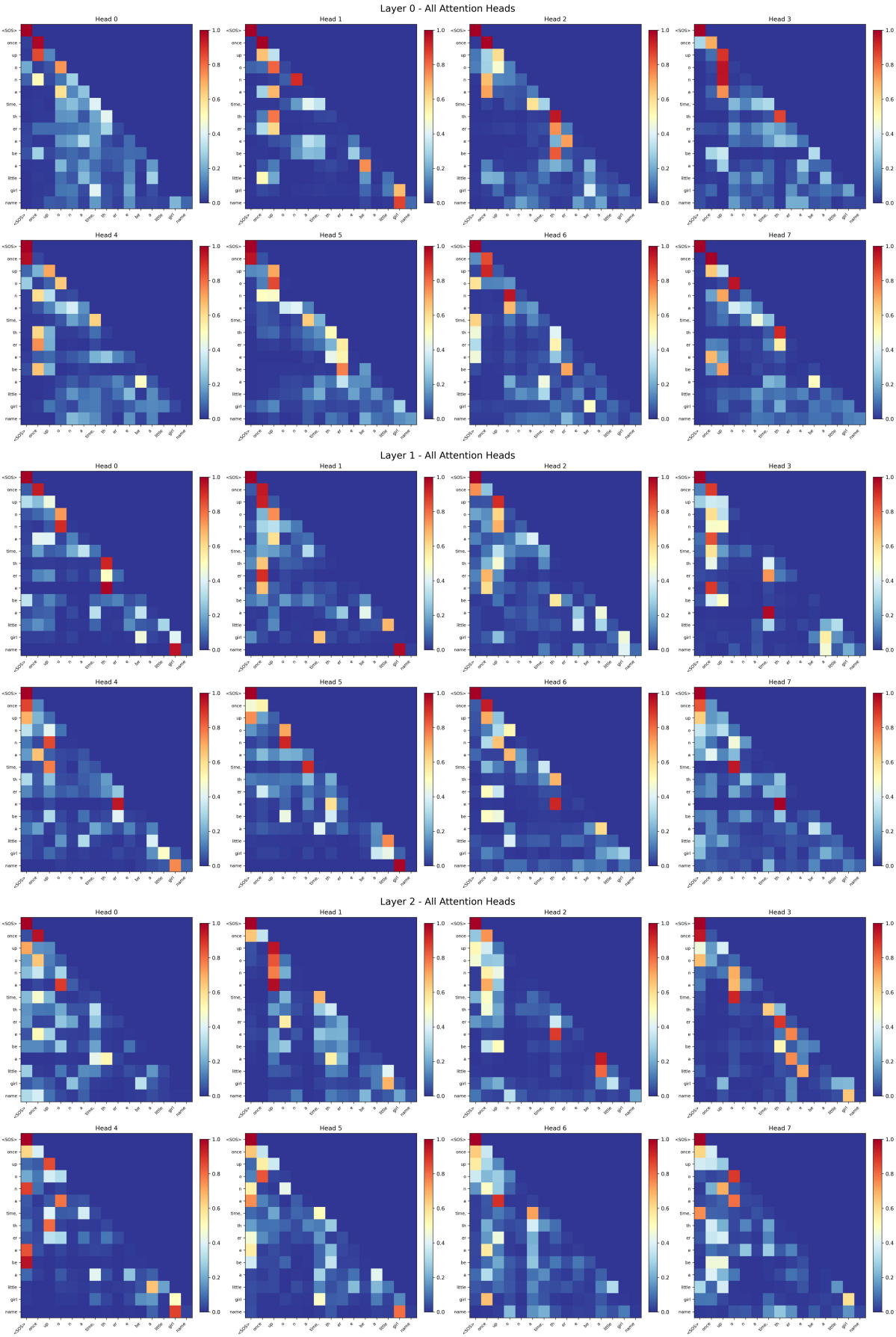

Figure 4: Attention head visualization for Example 2 (Layers 0–2).

cusing 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 shown in Table 5, increasing the beam width reduces runtime efficiency (172.5 $\rightarrow$ 1.6 tokens/sec) and yields lower BLEU scores. The degraded BLEU performance may be attributed to insufficient training epochs (due to computation), which limits the benefit from beam search .

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.

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

As shown, KV caching provides nearly a 2$\times$ increase in tokens per second compared to the baseline without caching

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.

As shown in Table 7, increasing the effective batch size via gradient accumulation leads to higher final training loss (from 2.5312 to 2.6080, a relative degradation of 1.2% to 3.0%) and increased runtime per epoch (191.60 s to 247.56 s at peak), with the exception of accumulation=8 showing slightly lower time, likely due to limited epochs.

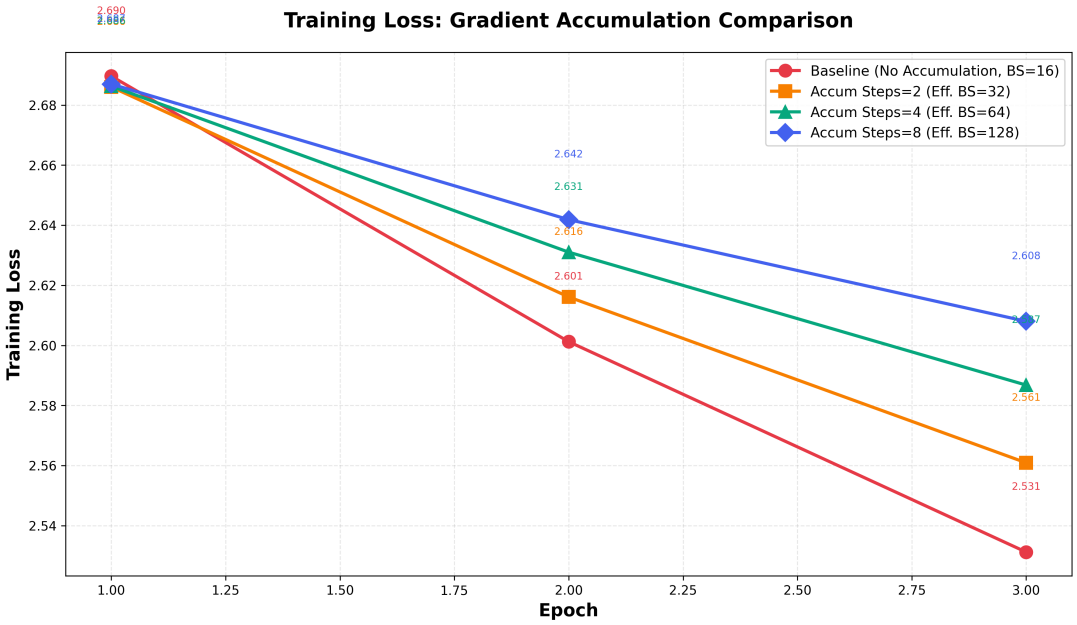

Figure 5: Training loss curves for baseline and accumulation steps 2, 4, 8.

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

4. Gradient Checkpointing

We implement manual gradient checkpointing for our transformer model. 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 (CPU-only).

Configuration	Final Loss	Avg Time/Epoch
Without Checkpointing	2.5283	242.74 s
With Checkpointing	2.5309	251.90 s

As shown in the table, manual gradient checkpointing on CPU slightly increases average time per epoch by **3.8%** (242.74 s $\rightarrow$ 251.90 s), consistent with expected recomputation overhead. Final training loss remains nearly unchanged (2.5283 vs. 2.5309), indicating no impact on convergence, benefiting from reduced memory.

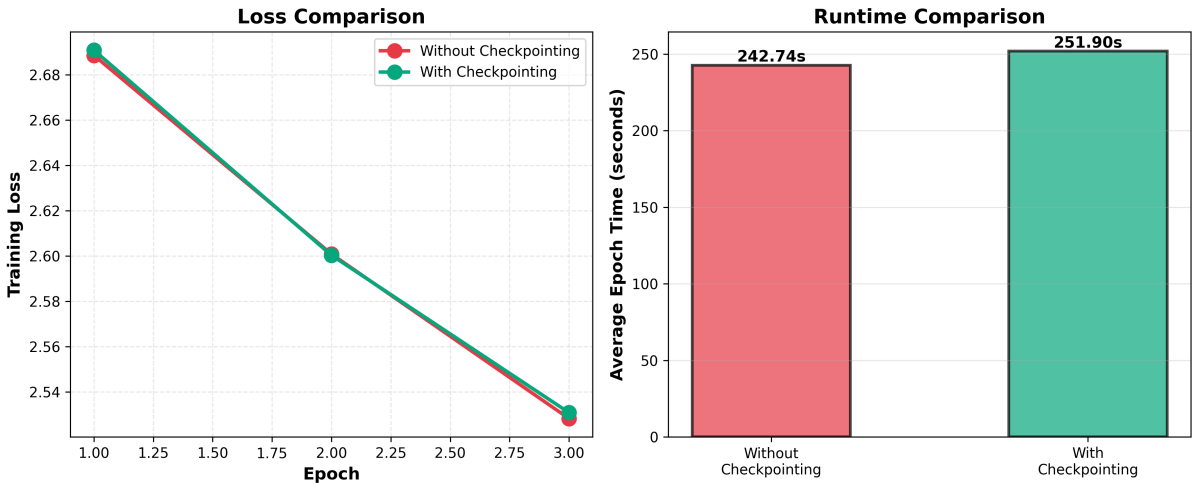

Figure 6: Loss and runtime per epoch with and without gradient checkpointing.