Decoder-Only Transformer Implementation
Assignment Report

Khushank Galgat — Entry Number: 2022MT11291

November 14, 2025

1 Introduction

The final trained model checkpoint is available at: `https://huggingface.co/koolkg26/customDecoder`
This report describes the implementation of a small decoder-only Transformer model trained on the TinyStories dataset. In this assignment, I builded all major components from scratch using PyTorch primitives (Linear, Embedding, LayerNorm, Dropout), loading FastText embeddings, implementing text cleaning, training the model, and performing multiple experiments including Beam Search, KV Caching, Gradient Accumulation, and Gradient Checkpointing.

2 Preprocessing

2.1 Mojibake Handling

The TinyStories dataset contains short synthetic stories. During inspection of the raw dataset, I observed the presence of **mojibake** (garbled Unicode characters). Common corrupted patterns included:

- "â€œ" instead of "
- "â€™" instead of ’
- "â€"" instead of —
- "Ã©" instead of é

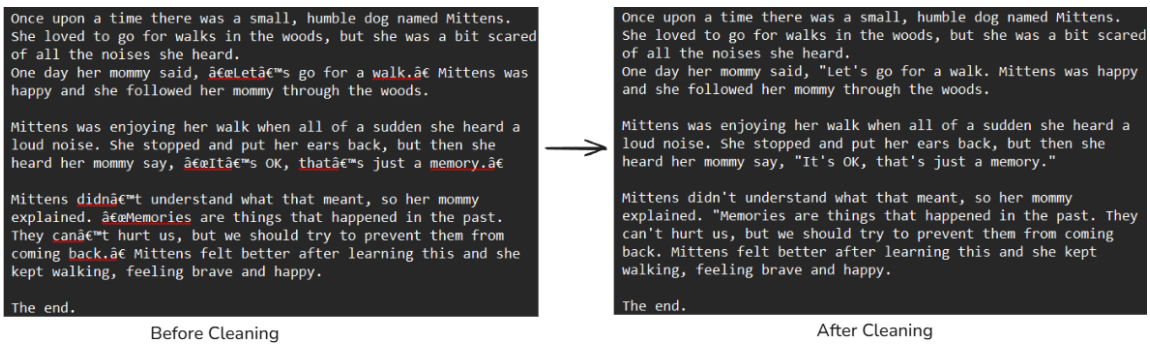

Figure 1: Example of mojibake present in the training dataset before cleaning.

To address this, I applied a two-step cleaning strategy:

1. Using the **ftfy** library to automatically fix Unicode and encoding-related artifacts.

2. Applying a **custom mojibake-normalization function** that performs targeted replacements for commonly corrupted characters such as:
    - fancy quotes → standard quotes (e.g., â€œ → “)
    - em dash / en dash artifacts → “–”
    - accented characters (Ã©, Ã±) → proper Unicode (é, ñ)
    - stray “Â” non-breaking space markers removed

This cleaning process greatly reduced noisy tokens and improved the coverage rate of pretrained FastText embeddings.

2.2 Tokenization

Tokenization was performed using:

- **spaCy** tokenizer (`en_core_web_sm`) with NER and parser disabled for efficiency.
- **Word-level** tokenization to ensure alignment with FastText pretrained vectors.

Special tokens used in the vocabulary:

- `[PAD]`
- `[UNK]`
- `[SOS]`
- `[EOS]`

These tokens were appended to the vocabulary and integrated into the embedding matrix.

3 Model Architecture

3.1 Vocabulary and Embeddings

To build the vocabulary, I tokenized **both the training and validation sets** using the spaCy word-level tokenizer. This ensured that all unique words appearing in either split were included in the vocabulary.

The final vocabulary size after preprocessing was:

$$\text{Vocab size} = 56{,}333 \text{ tokens}$$

I used pretrained **FastText 300-dimensional English embeddings**. Out of the total 56,333 tokens:

- Approximately **48,000 tokens** were already present in FastText.
- The remaining were treated as **OOV (Out-of-Vocabulary)**.

For embedding construction:

- I used FastText's `get_word_vector()` method.
- This returns a valid 300-dimensional vector for *every possible input string*— meaning it generates vectors for both known and OOV words.

Thus, the final embedding matrix shape becomes:

$$56337 \times 300$$

This includes four additional special tokens: `[PAD]`, `[UNK]`, `[SOS]`, `[EOS]`.

3.2 Model Structure

The model is a decoder-only Transformer consisting of 3 stacked decoder blocks. Each block follows:

- LayerNorm
- Multi-Head Self-Attention
- LayerNorm
- Feed-Forward Network

```
<bound method Module.parameters of DecoderOnlyTransformer(
  (token_embedding): Embedding(56337, 300)
  (layers): ModuleList(
    (0-2): 3 x DecoderLayer(
      (self_attn): MultiHeadSelfAttention(
        (wq): Linear(in_features=300, out_features=300, bias=False)
        (wk): Linear(in_features=300, out_features=300, bias=False)
        (wv): Linear(in_features=300, out_features=300, bias=False)
        (wo): Linear(in_features=300, out_features=300, bias=False)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (ln1): LayerNorm()
      (ff): FeedForward(
        (fc1): Linear(in_features=300, out_features=1200, bias=True)
        (fc2): Linear(in_features=1200, out_features=300, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (activation): ReLU()
      )
      (ln2): LayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (final_ln): LayerNorm()
  (output_linear): Linear(in_features=300, out_features=56337, bias=False)
)>
```

Figure 2: Model parameter summary (`model.parameters()` output).

```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadSelfAttention(d_model, num_heads, dropout)
        self.ln1 = LayerNorm(d_model)
        self.ff = FeedForward(d_model, hidden_dim=4 * d_model, dropout=dropout)
        self.ln2 = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, key_padding_mask=None,output_attentions=False):
        # Self-attention (with pre-LN)
        x_norm = self.ln1(x)
        attn_out, attn_weights = self.self_attn(x_norm, padding_mask=key_padding_mask)
        x = x + self.dropout(attn_out)

        x_norm2 = self.ln2(x)
        ff_out = self.ff(x_norm2)
        x = x + self.dropout(ff_out)
        return x,attn_weights
```

Figure 3: DecoderLayer forward pass showing pre-LN design (LayerNorm → Attention).

3.3 Baseline Configuration

The baseline model configuration used for all the main experiments is:

- Context length: **64**
- Number of decoder layers: **3**
- Attention heads: **5**
- Embedding (hidden) dimension: **300**
- Feed-Forward (inner) dimension: **1200**
- Dropout: **0.1**
- Optimizer: **Adam** (`lr = 3e-4`, no weight decay)
- Hardware: **2 × NVIDIA T4 GPUs** (Kaggle : Used DDP)
- Total parameters: **37,050,900**
- Trainable parameters: **20,149,800**

Freezing the Embedding Layer: The embedding layer was kept *frozen* throughout training. Since I constructed a complete vocabulary from both the training and validation sets and performed thorough mojibake cleaning, almost all tokens had clean representations. Furthermore, FastText provides high-quality, pretrained 300-dimensional word vectors that already capture rich semantic structure.

For these reasons, the FastText embedding matrix was considered sufficiently robust, and fine-tuning it was unnecessary. Freezing the embeddings also reduced trainable parameters and improved training stability.

4 Training

Training was performed using teacher forcing with cross-entropy loss computed over the shifted target sequence. Training was performed on full dataset and took 1 hr 45mins approx for each epoch. I ran for 10 epochs.

4.1 Training and Validation Loss

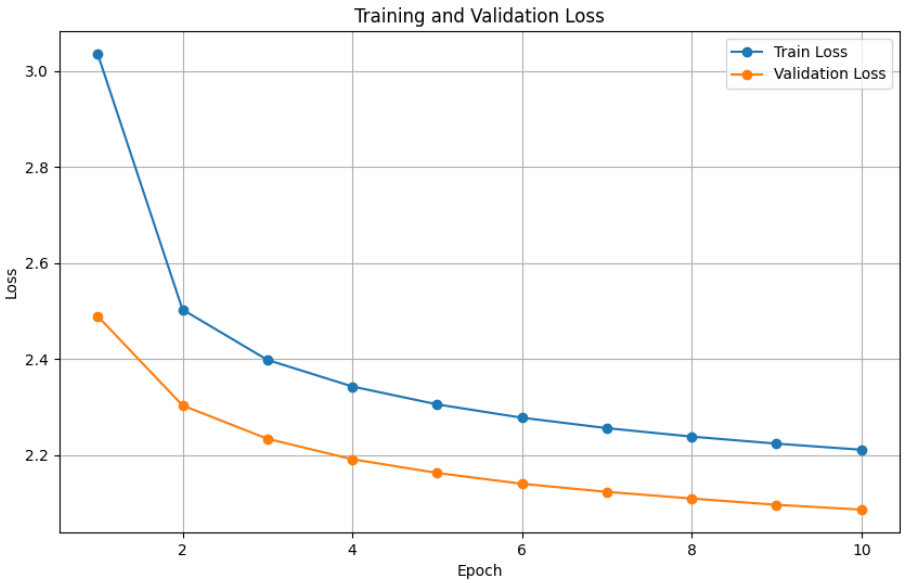

Figure 4: Training and validation loss across epochs.

4.2 Perplexity vs Epoch

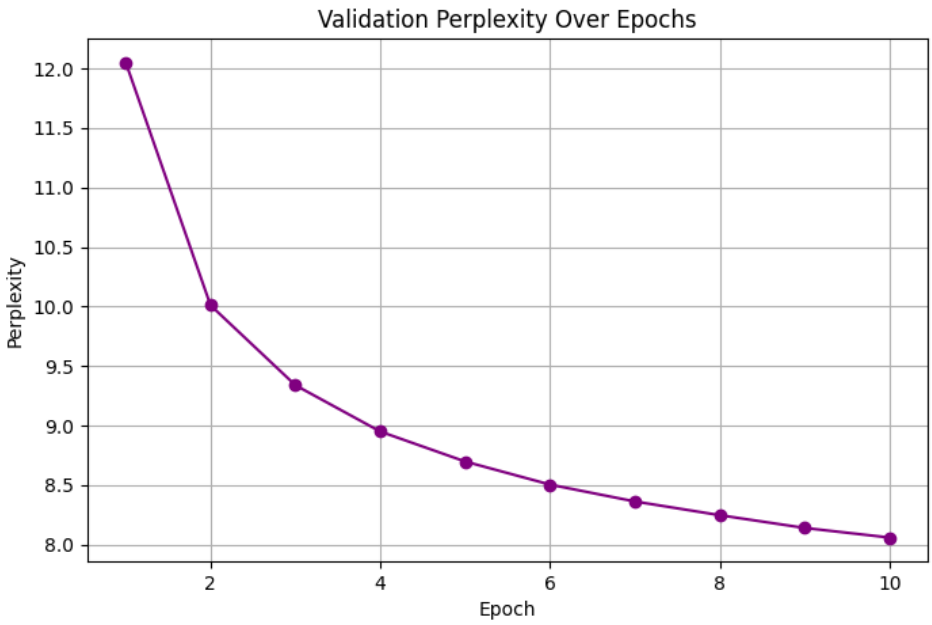

Figure 5: Perplexity trend across epochs.

5 Inference

5.1 Stochastic Sampling in Generation

For generating continuations, the `generate()` function was implemented using **Top-k sampling** and **temperature scaling**. Throughout most experiments, the generation settings were:

- Top-k: **50**
- Temperature: **0.85**

These parameters introduce controlled randomness while maintaining coherence, which is suitable for creative text such as TinyStories.

5.2 Evaluation on Validation Samples

I sampled **50 sentences** from the validation split. For each sample, only the **first 5 tokens** were used as the prompt, and the model generated a continuation.

The following metrics were computed over these 50 generated continuations:

- **Average Per-Token Perplexity**: **4.219**
- **Average BLEU Score**: **0.0020**

Discussion: The BLEU score is extremely low, but this is expected for open-ended story generation. TinyStories is a creative dataset, and for a given prompt, there are countless valid continuations. BLEU penalizes lexical variation heavily and thus is not a suitable metric for creative narrative text generation. Perplexity is a much more meaningful indicator of model quality in this setting.

5.3 Attention Head Visualizations

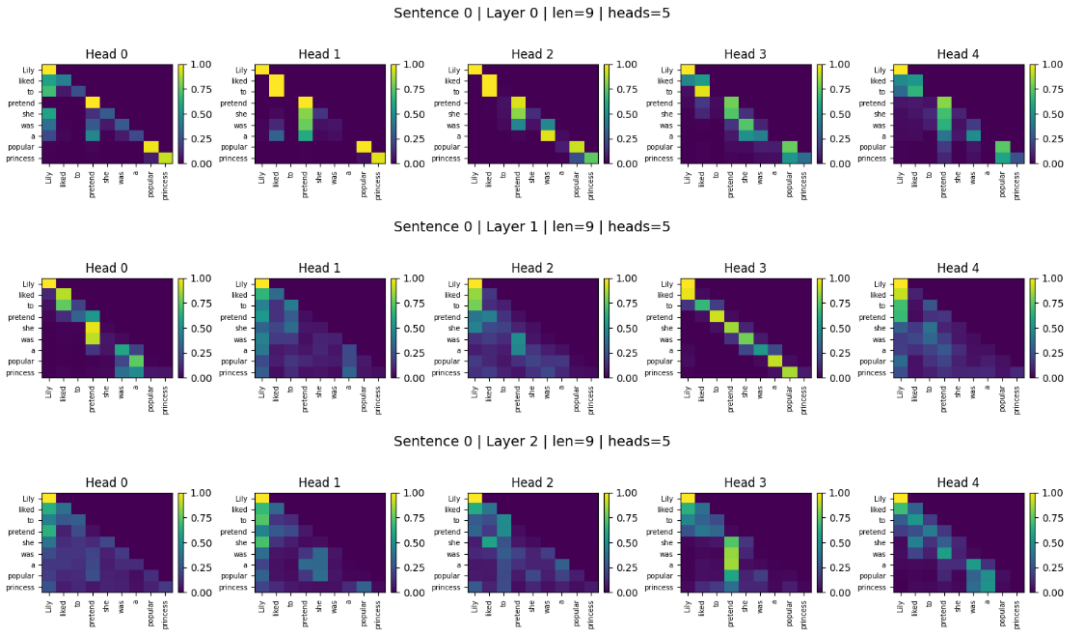

Figure 6: Visualization of attention heads across layers for sentence : Lily liked to pretend she was a popular princess.

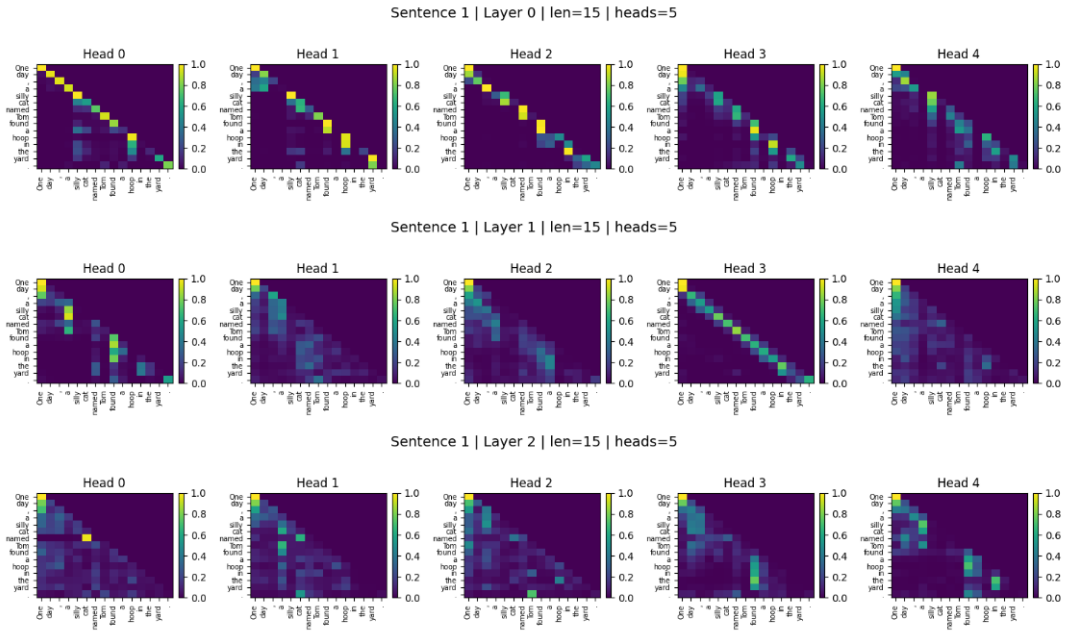

Figure 7: Visualization of attention heads across layers for sentence : One day, a silly cat named Tom found a hoop in the yard.

Discussion For the shorter sentence about "Lily", the attention maps across all layers remain strongly diagonal, indicating the model relies almost entirely on short-range context. While *Layer 2, Head 3* shows mild long-range links from "pretend" toward "princess", no head meaningfully connects the pronoun "she" with the predicate phrase "a popular princess". This exposes a limitation of the current architecture (3 layers, 5 heads): it struggles to form deeper coreference or role-binding relationships, suggesting that additional layers or more heads would likely be needed for richer semantic reasoning.

In contrast, for the longer sentence about "Tom the cat", some heads—most notably in *Layer 2*—begin to exhibit broader attention windows, linking action tokens such as "found" and "hoop" back to the subject phrase ("a silly cat named Tom"). This shows encouraging signs of emerging agent–action coherence. However, even here, no head successfully binds the final location phrase ("in the yard") back to the action, and attention remains mostly local and diagonal. This indicates that while the model begins to capture longer-range dependencies in longer sentences, it still lacks the capacity to construct complete event structures (agent–action–object–location) due to its small size.

6 Training and Inference Enhancements

6.1 Beam Search Decoding

Beam search was evaluated with beam widths $k = 5$ and $k = 10$ and compared against stochastic sampling (top-$k = 50$, temperature 0.85). Figure 8 shows the decoding speed and BLEU scores for each method.

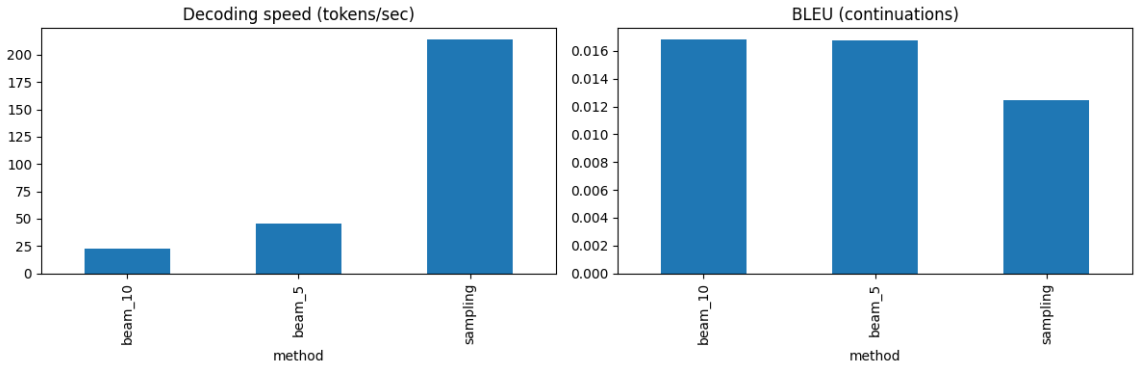

Figure 8: Comparison of decoding speed (tokens/sec) and BLEU quality across sampling, beam search $k = 5$, and $k = 10$.

Speed–Quality Trade-off: Sampling is extremely fast ($\sim$ 210 tokens/sec) but produces the lowest BLEU. Beam search improves coherence significantly but is far slower (especially at $k = 10$), with diminishing returns between $k = 5$ and $k = 10$. This reflects the classic trade-off between exploration breadth and computational cost.

Output Examples

Prompt 1: `Once, there was a boy named Tim.`

Generated by Sampling (top-$k = 50$, temp $= 0.85$):

> *Tim was very scared of the dark. He did not want to play with his friend, Tim. Tim tried to say, "Please, Tim, let me play with my friend." Tim said, "Okay, Tim. I will be careful."*

Generated by Beam Search ($k = 5$):

> *Tim had a toy car that he loved very much. He played with it every day and had lots of fun. One day, Tim's friend, Sam, came over to play. Sam saw Tim's toy car and wanted to play with it too.*

Observations: Sampling degenerates into repetition and pronoun/name looping ("Tim ... Tim ... Tim"), showing unstable semantic grounding. Beam search ($k = 5$) maintains topic consistency: a toy, a friend, a simple event—aligned with TinyStories' narrative structure.

Prompt 2: `Once upon a time, there was an island`

Generated by Sampling:

> *in the middle of the sea. It was so full of shiny coins and shiny things. Suddenly, a big gust of wind blew the oyster away. The oyster was so happy to see the island and...*

Generated by Beam Search ($k = 5$):

> *The island was very big and full of people. One day, a little girl came to the island. She saw the island and wanted to explore it. She asked her mom if she could go...*

Observations: Sampling introduces random objects ("shiny coins", "oyster") and drifts rapidly away from the prompt's theme. Beam search maintains narrative grounding and produces a stable, child-friendly continuation that matches TinyStories' style.

Overall Output Quality Summary: Beam search ($k = 5$) provides far more coherent, structured, and prompt-aligned stories. Sampling is faster and more creative but suffers from repetition, topic drift, and inconsistency. For this small 3-layer model, beam search strikes the best balance between quality and stability.

6.2 KV Caching

To measure the impact of Key–Value (KV) caching, I evaluated **20 validation samples**. For each sample:

- Top-k = **50**
- Temperature = **1.0**
- Maximum new tokens = **50**

The speed comparison is shown in Table 1.

Mode	Tokens Generated	Time (s)	Tokens/sec
No KV Cache	867	4.464	194.2
With KV Cache	948	3.687	257.1

Table 1: KV Caching Speed Comparison

Discussion: KV caching provides a clear speedup: from **194 → 257 tokens/sec**, approximately a **32% improvement**. This is expected because during autoregressive decoding, previously computed K and V tensors do not need to be recomputed at every timestep. For longer sequences, the speedup would be even more significant because the cost of repeated self-attention grows with sequence length.

6.3 Gradient Accumulation

Gradient accumulation was used to simulate larger effective batch sizes while staying within Kaggle's GPU-hour constraints. A fixed mini-batch size of **16** was used, and experiments were run on **2×T4 GPUs**. To make multiple runs feasible, training was conducted on **10% of the training dataset**. This reduced runtime significantly while still preserving the comparative behavior of different accumulation settings.

Accumulation steps tested were 1, 2, 4, 8.

Accumulation Steps	Avg. Epoch Time (s)	Effective Batch Size
1	10 min 13 sec	$1 \times 16 \times 2 = 32$
2	9 min 25 sec	$2 \times 16 \times 2 = 64$
4	8 min 55 sec	$4 \times 16 \times 2 = 128$
8	8 min 45 sec	$8 \times 16 \times 2 = 256$

Table 2: Epoch Runtime and Effective Batch Sizes for Gradient Accumulation

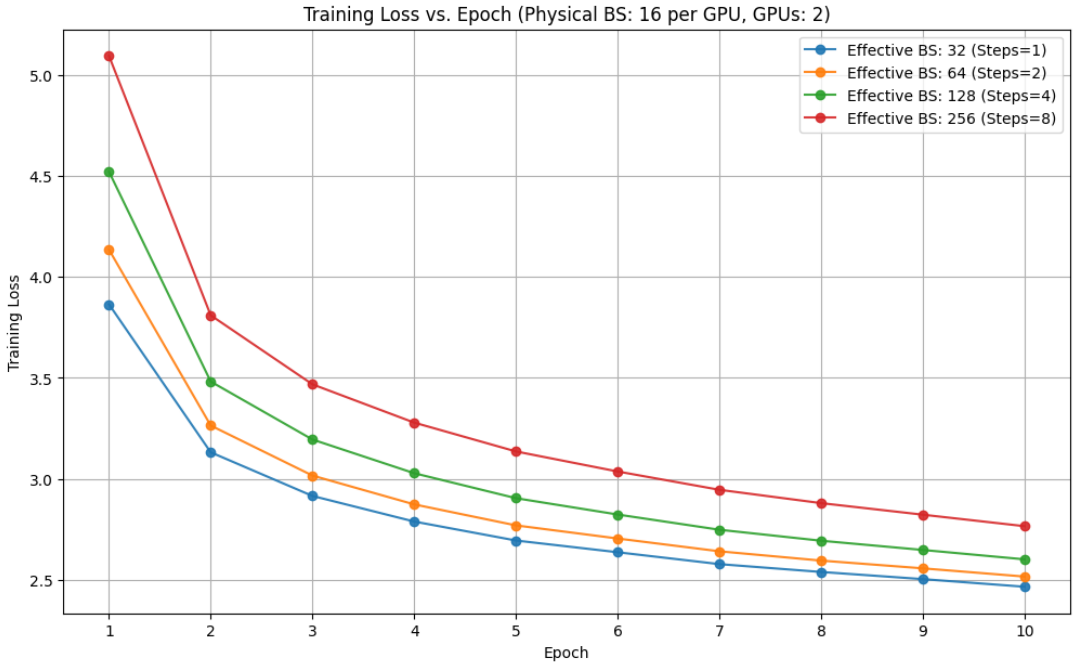

Figure 9: Training loss curves for accumulation steps 1, 2, 4, and 8.

Discussion: As the effective batch size increases, the epoch time is actually seen *decreasing*, which is not unintuitive. With larger accumulation steps, the optimizer is called far less frequently, and each optimizer step is one of the most expensive operations in training—especially in a 2-GPU setup where every update triggers cross-GPU synchronization. Reducing the number of these updates naturally shortens the overall training time. Moreover, because the model is small (3 decoder layers) and the experiment uses only 10% of the dataset, the forward/backward compute per microbatch is cheap, while the real cost comes from fixed overheads such as DDP communication, Python loop iterations. Therefore, it makes sense that with larger effective batch sizes, the epoch time decreases rather than increases.

6.4 Gradient Checkpointing

To reduce GPU memory consumption, I implemented **manual gradient checkpointing** by wrapping each DecoderLayer inside a custom `torch.autograd.Function`. During the forward pass, this wrapper:

- executes the DecoderLayer inside a `torch.no_grad()` context,
- preventing PyTorch from storing intermediate activations.

During the backward pass, the wrapper:

- re-runs the forward pass (this time tracking gradients),
- regenerates the needed activations,
- and performs the gradient computation.

This trades additional computation for significantly reduced VRAM usage.

Setting	Peak GPU Memory (GB)	Epoch Time (s)
No Checkpointing	6.08	1 hr 44 min 48 sec
With Checkpointing	5.56	1 hr 48 min 59 sec

Table 3: Gradient Checkpointing Memory and Runtime Comparison

Discussion: Gradient checkpointing reduces peak GPU memory usage from **6.08 GB** to **5.56 GB**, which is expected because activations are not stored during the forward pass and must be recomputed during the backward pass. However, the epoch time becomes *slightly slower* when checkpointing is enabled (1 hr 48 min 59 sec compared to 1 hr 44 min 48 sec). This is consistent with the typical trade-off of checkpointing: by saving memory, the model must redo part of the forward computation during backpropagation, adding extra compute overhead. Since the model here is relatively small, the recomputation cost is noticeable, while the memory savings do not translate into a speed benefit. Thus, the result clearly reflects the standard behavior of gradient checkpointing: **lower memory usage in exchange for increased runtime due to recomputation**.

7 Sample Generations

Here are five example continuations generated by the model using short prompts taken from the validation split. These were selected because they show comparatively stronger coherence and narrative structure.

Prompt	**Generated Continuation**
`One day, Tommy and`	*his mom were on a walk. As they walked, Tommy saw a big, shiny ball. He was so excited to play with it, but his mom said they had to be careful. Timmy asked, "Mom, can..."*
`Once, Mrs. Miller had`	*an idea. She found a long stick near her house. She used it to attach her sticks to the leaves. After a few minutes, Mrs. Miller showed Lily a picture of a snowman. Lily was so happy. She asked...*
`One day, a yellow`	*bird flew overhead in the sky. "What's that?" Amy asked. She pointed at the sky and said, "That's a comet, Amy. It can help it grow." Amy said, "Wow,..."*
`Alice was hungry. She`	*asked her mom for some toast. Her mom gave her a big plate of toast and a spoon. Alice was so happy and ate the whole toast quickly. After that, Alice was tired but happy. She had a great time...*
`John was three years old`	*and he was so excited to go inside. He grabbed his dad's hand and ran outside. John's dad said, "Wait! Let's go inside!" John was scared. But his dad said, "Okay,..."*

Table 4: Five higher-quality sampled generations selected from the validation prompts.