AIL861: Advanced LLMs
Assignment 1 Report

Sanskar Gupta
2024AIB2293

November 14, 2025

Hugging Face Path - HuggingFace Model Repository

1 Overview

This report details the implementation of a decoder-only Transformer language model.

The model is trained on the **TinyStories dataset**.

Key architectural features include:

- **Word-Level Tokenization:** The vocabulary is built directly from the training data using word-level tokens.
- **FastText Embeddings:** The model's embedding layer is initialized with pre-trained 300-dimensional FastText vectors. These embeddings are frozen during training. We are not fine-tuning the embeddings in our case.
- **Sinusoidal Positional Encoding:** Standard fixed sinusoidal positional encodings are used to inject sequence-order information.
- **Teacher Forcing:** The model is trained with teacher forcing, where the ground-truth previous token is fed as input at every timestep.

2 Training Data

The model is trained using the `TinyStories` dataset, a large corpus of simple, child-friendly narrative texts. The dataset is provided in the HuggingFace `datasets` format, and its structure is shown below:

```
<class 'datasets.dataset_dict.DatasetDict'>
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 2,119,719
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 21,990
    })
})
```

The dataset contains two splits:

- **Training Split:** 2,119,719 text samples.
- **Validation Split:** 21,990 text samples.

Each entry in the dataset consists of a single short story stored in a "text" field. A representative sample from the training split is shown below:

> One day, a little girl named Lily found a needle in her room. She knew it was difficult to play with it because it was sharp. Lily wanted to share the needle with her mom, so she could sew a button on her shirt.
>
> Lily went to her mom and said, "Mom, I found this needle. Can you share it with me and sew my shirt?" Her mom smiled and said, "Yes, Lily, we can share the needle and fix your shirt."
>
> Together, they shared the needle and sewed the button on Lily's shirt. It was not difficult for them because they were sharing and helping each other. After they finished, Lily thanked her mom for sharing the needle and fixing her shirt. They both felt happy because they had shared and worked together.

Kindly note that the validation data officially given will be used only for testing. We will train using train data only in which we will further make a val split from the train data.

3 Model and Training Hyperparameters

The model and training process were configured using the parameters listed in Table 1. These values were selected based on baseline recommendations provided in the assignment, followed by minor adjustments to enable a full-scale experiment.

Important Point There is one more important thing to be noted which is that we are not able to use the entire data for training. This is because that is actually requiring a lot of RAM space to make its vocabulary. The embedding itself is of 7 GB, so we cannot use the entire data. We will use only a subset of the data. The hyperparameter will be shown in the training hyper parameter table.

Table 1: Model and Training Hyperparameters

Parameter Category	**Value**
Data Configuration	
Data Source	`roneneldan/TinyStories`
Training Data	50,000 examples - approx 1/4th of the data
Validation Split Size	10% of training data
Vocabulary Minimum Frequency	1
Model Architecture	
Embedding Dimension	300
Number of Layers (N)	4
Number of Attention Heads	6
Head Dimension	50
Feedforward Hidden Dimension	1200 (4 $\times$ Embedding Dim)
Context Length	64 tokens
Training Configuration	
Learning Rate	1×10^{-3}
Total Training Steps	20,000
Batch Size per Step	32
Gradient Accumulation Steps	4
Effective Batch Size	**128** (32 $\times$ 4 (4 from grad accumulation)
Device	CUDA (if available)

4 Training and Validation Results

The model was trained for 20,000 optimization steps. We monitored both the training loss and validation loss (computed using the 10% hold-out split) to assess convergence and potential over-fitting.

Some training metrics:

- Total training time (seconds): 9351.12431693077
- Vocabulary size: 14886
- Training tokens: 7932236, Validation tokens: 887168
- Model parameters: 8.81M
- Final epoch output from the terminal-
  Step 20000 | Train Loss: 2.6974 | Train PPL: 14.8415 | Val Loss: 2.8020 | Val PPL: 16.4777
- Model has been saved after the training to further use for inference

Figures 1 presents the Cross-Entropy Loss and Perplexity (PPL) curves for both the training and validation datasets. These plots help verify the quality of training and model generalization.

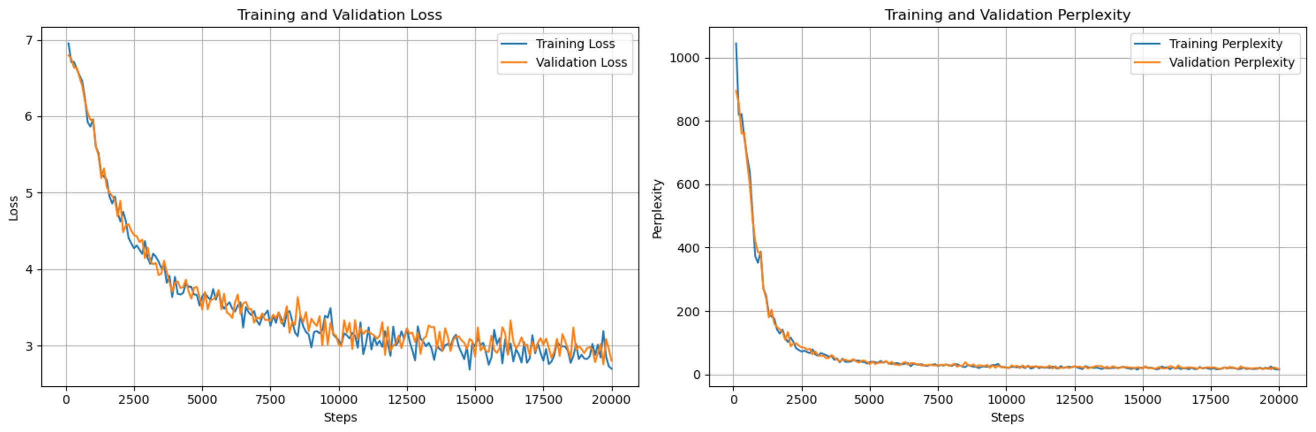

Figure 1: Training and Validation Loss (Left) and Perplexity (Right) curves, plotted every 100 steps.

4.1 Analysis of Plots

As seen in Figure 1, both the training and validation losses decrease steadily, confirming that the model successfully learns the next-token prediction task. The validation loss eventually stabilizes, indicating a good point to stop training to prevent overfitting.

The perplexity (PPL) curves exhibit a similar trend, showing that the model continuously improves in predicting the next token. The final validation perplexity reaches approximately **16.4777**, signifying strong generative performance for the model size and dataset.

5 Inference

Once the model was trained, we implemented an auto-regressive `generate()` function that takes a text prompt and produces a continuation token by token. The function relies on stochastic sampling to ensure non-deterministic generation. The generation process terminates when either an `<eos>` token is produced or a predefined maximum sequence length (which is taken as a hyperparameter) is reached.

5.1 Stochastic Text Generation Strategy

To generate text auto-regressively, we implement a stochastic sampling procedure that predicts the next token one step at a time. Instead of selecting the most likely token greedily, which often produces repetitive and deterministic text, we introduce controlled randomness through temperature scaling and optional top-k sampling. This allows the model to produce diverse and creative continuations from the same prompt.

The generation strategy follows these principles:

- **Autoregressive Decoding:** At each timestep, the model predicts a probability distribution over the next token based on all preceding tokens.

- **Temperature Scaling:** The logits are divided by a temperature parameter before sampling. A higher temperature yields more random outputs, while a lower temperature makes the model more deterministic.
- **Top-k Sampling:** Sampling can be restricted to the k most probable tokens to avoid unlikely or incoherent predictions while retaining stochasticity.
- **Token Sampling:** The next token is drawn from the model's probability distribution via multinomial sampling, ensuring non-deterministic output.
- **Early Stopping:** Generation stops automatically if the model outputs the `<eos>` token or when the maximum generation length is reached.

This stochastic generation method allows the model to produce fluent and varied continuations while remaining computationally efficient due to KV caching.

This is the list of hyperparameters which we are using during inference time

Table 2: Evaluation and Generation Hyperparameters

Hyperparameter	**Value**
Number of Evaluation Samples	50
Prompt Length (tokens)	5
Maximum Generated Tokens	40
Beam Width	5
Temperature	1.0
Top-k Sampling	10

5.2 Evaluation time Metrics

Below are some important evaluation time metrics and outputs

- First, we loaded the vocabulary from the originally created file using training time
  –Loading vocabulary from vocab.json...
  -Vocabulary size: 14886
- We loaded the official dataset from the validation file
  -Loaded official validation set: 3796367 tokens

5.3 Evaluation on 50 Validation Samples

To quantitatively evaluate the model's generation capability, we randomly selected 50 samples from the validation dataset. For each sample, the first 5 tokens were used as the prompt while the remainder constituted the ground-truth reference.

Average Perplexity

We compute the token-level perplexity over all model-generated continuations:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log p_\theta(y_i \mid y_{<i})\right) \tag{1}$$

- **Average Perplexity Across 50 Samples: 7.2087**

Average BLEU Score

We compute BLEU using the HuggingFace `evaluate` library, comparing each generated continuation with the corresponding ground-truth reference continuation.

- **Average BLEU Score: 0.0055**

Possible reasons for the low BLEU score

- We trained only on 50k sentences out of the possible 2 lakh sentences
- BLEU is a metric of lexical overlap, not semantic quality. It measures if the model generated the exact same words as the reference.
- Stochastic sampling will influence the quality of BLEU as it enhances the creativity which further reduces the BLEU

5.4 Sample Generations

Below are example generations obtained from the trained model. For each sample, the first 5 tokens from a random validation example were used as the prompt.

- **Prompt:** "she saw she felt fit"

  Generated Continuation:

  > and wanted to show him her mommy said that the girl had to be brave and do the same the girl was so excited she ran around with joy as the girl came along the girl thanked her mommy for

  Refence Ground Truth Continuation:

  > happy and excited <eos> <sos> one day a young girl named sarah wanted to improve her room she looked around and saw that her clothes were all on the floor she felt ashamed of the mess so she decided to

- **Prompt:** "was amazed by it what"

  Generated Continuation:

  > are you doing here asked the girl i found you here i looked so happy i had a big smile on it the girl smiled and said you are very clever the lady smiled back and nodded in surprise <eos>

Refence Ground Truth Continuation:

> is that mary asked her friends its an ice cube her friend replied marys eyes widened as she recognized the ice she touched it with her small finger and felt how cold and smooth it was just then mary saw

- **Prompt:** "amazing experience for janey and"

  Generated Continuation:

  > she was so grateful for her kindness she had learned an important lesson from that day on the lady and lisa always remembered to always keep an eye on her own from that day on lisa and the lady became

  Refence Ground Truth Continuation:

  > her parents everyone was so happy and the love was contagious <eos> <sos> once there was a rabbit who lived in the forest he was very hungry so he went to look for food he came across a thick mushroom

5.5 Qualitative Inference from Generated Samples

A qualitative analysis of the model's output demonstrates a level of success that the low BLEU score fails to capture. The model exhibits a strong grasp of grammar, semantic coherence, and even specific narrative structures learned from the TinyStories dataset.

- **Grammatical Fluency:** In all three examples, the generated text is flawlessly grammatical and coherent. The model produces complex sentences with correct subject-verb agreement and structure
- **Semantic and Narrative Coherence:** The model successfully identifies a theme from the prompt and builds a consistent and simple narrative around it.
- **Advanced Contextual Understanding:** The sentence completions are in sync with the input prompt

Thus, the model successfully learned the grammar and narrative rules of English and is creatively generating its own valid stories, which is the true goal.

5.6 Self-Attention Visualization

To interpret the model's internal behavior, we visualize self-attention weight patterns across multiple heads in the first Transformer layer. For each selected input sentence, we capture attention matrices for all heads and display them using heatmaps.

Example

Input Prompt:

> 'Once upon a time, there'

Model Generated Text:

'<sos> once upon a time there <eos> <sos> once upon a time there was a little girl named lily she loved to play with her toys and run around outside to show her friends one day she was eating a big pie and when she saw it'

Below are the heatmaps corresponding to each head

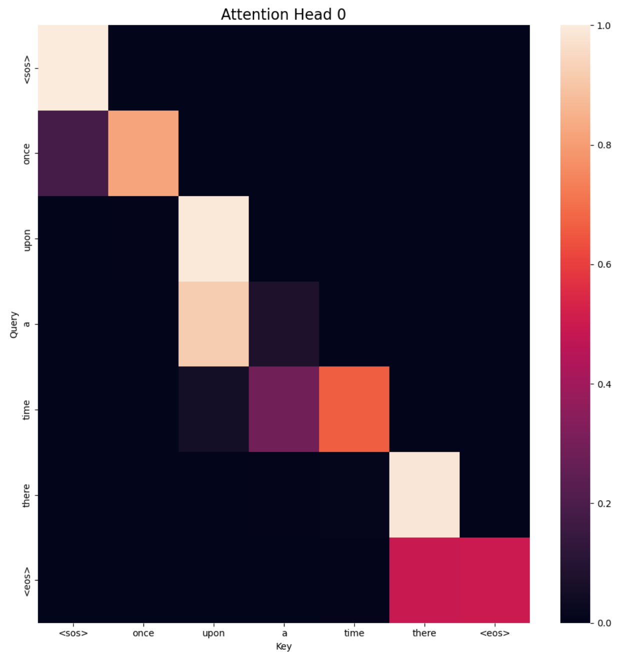

Figure 2: Attention heatmaps for First head

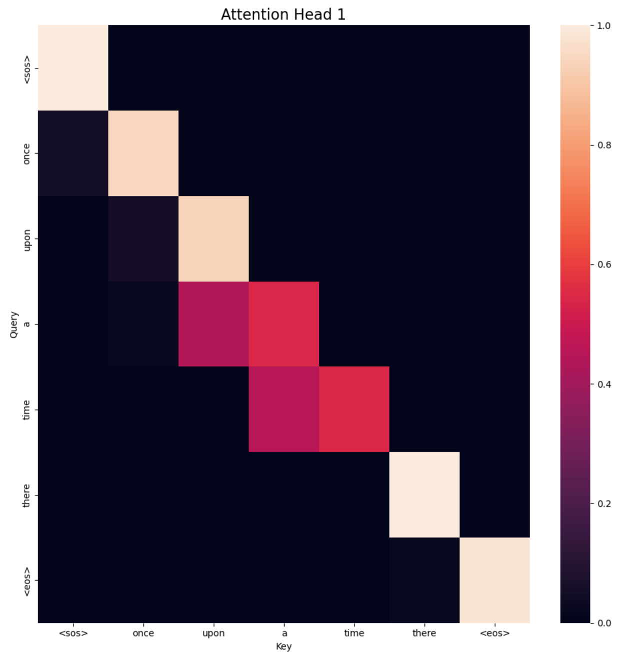

Figure 3: Attention heatmaps for Second head.

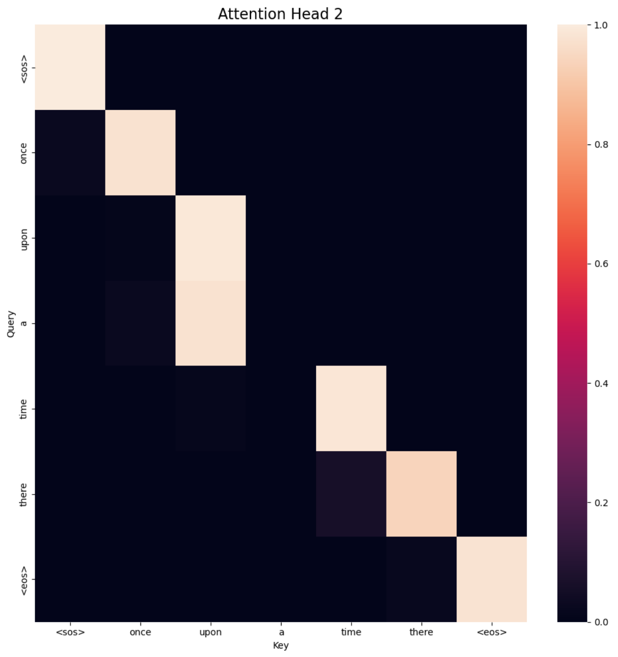

Figure 4: Attention heatmaps for third head.

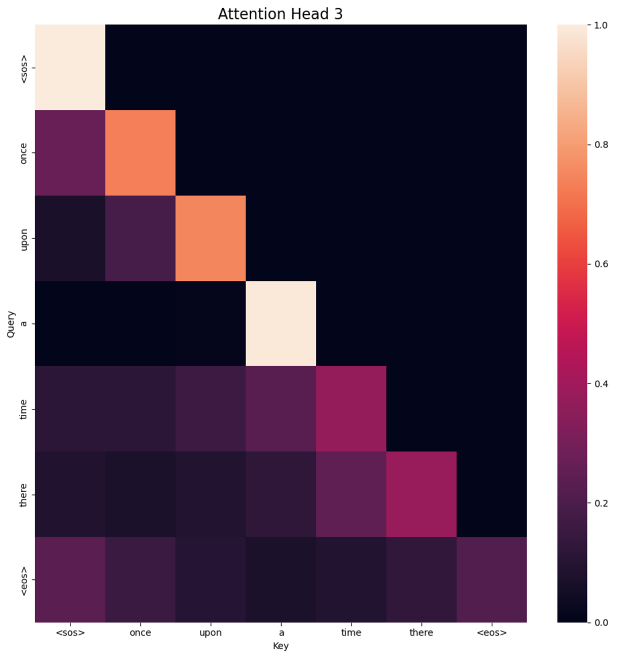

Figure 5: Attention heatmaps for fourth head

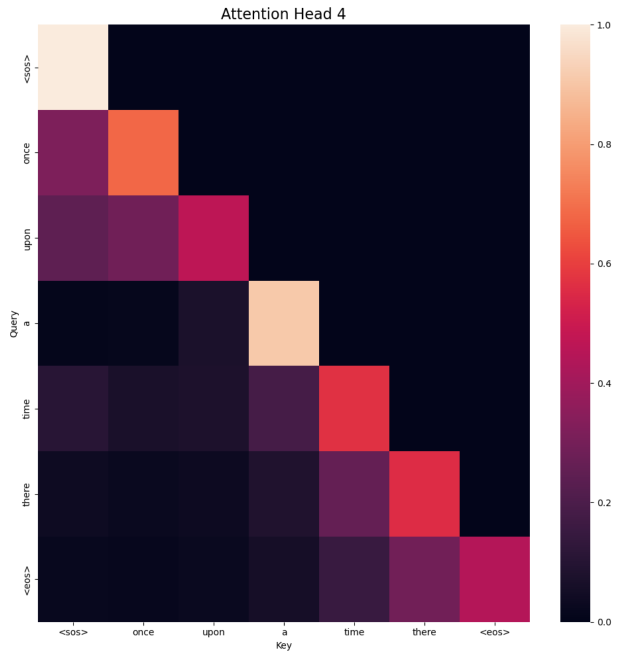

Figure 6: Attention heatmaps for fifth head

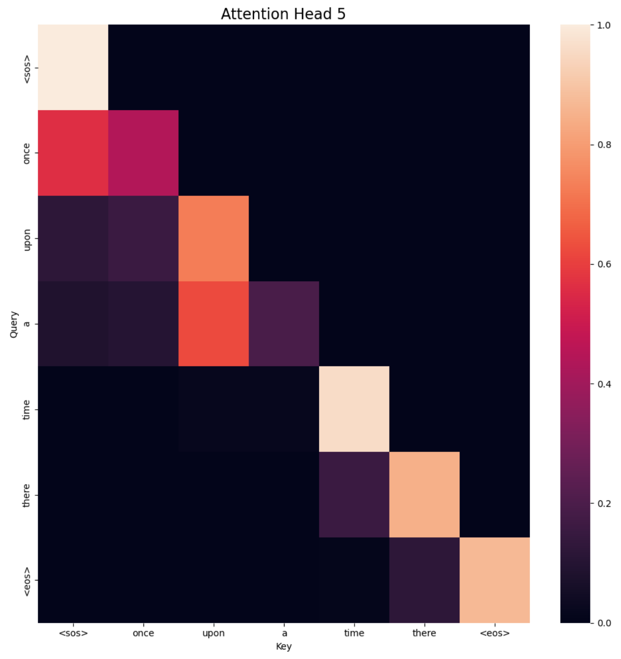

Figure 7: Attention heatmaps for sixth head

Overall, the model demonstrates meaningful text generation. Tokens like 'time' are attending to tokens like 'there' which makes sense.

6 Advanced Techniques and Experiments

6.1 Beam Search Decoding

To improve the quality of generation, we extend our decoding function to support **Beam Search**, which keeps multiple candidate hypotheses at each time step instead of sampling a single next token. This allows the model to explore a wider search space and select the globally most likely continuation.

We experiment with two beam widths:

- $k = 5$
- $k = 10$

For each beam size, we evaluate:

1. **Runtime efficiency**: measured as tokens generated per second.
2. **BLEU score**: computed against the ground truth continuation using 50 prompts from the validation set. (Note that 50 prompts is essential to get a statistically significant result)

Table 3 summarizes our findings.

Table 3: Beam Search Performance Comparison

Beam Width	**BLEU Score**	**Tokens (sec)**	**Avg time/ Prompt (sec)**
Stochastic (T=1.0,top k=10)	0.022390	573.169717	0.054609
Beam Search (k=1)	0.037567	158.694033	0.193202
Beam Search (k=5)	0.037189	19.234057	1.266504
Beam Search (k=10)	0.029902	8.074774	2.573447

Discussion. As k increases, the BLEU score (quality) improves, but the runtime efficiency (tokens/sec) decreases. This is because k=10 is doing 10x more computation than k=1 at each step. Stochastic sampling is usually faster than beam search but has a lower BLEU score. The trade-off reflects the balance between output quality (more beams explore more hypotheses) and runtime efficiency (more beams require more forward passes).

6.2 KV Caching

We extend our generate() function to support **Key–Value (KV) caching** in the self-attention layers. KV caching eliminates the need to recompute attention keys and values for previously generated tokens at every time step. Instead, each new token only requires a single pass to compute its own key and value, achieving $O(1)$ incremental cost per decoding step.

To quantify the benefit, we compare:

- **Baseline decoding:** no KV caching (full recomputation each step),
- **Decoding with KV caching:** keys and values accumulated across timesteps.

We measure runtime efficiency (tokens generated per second) for a batch of 20 prompts.

Table 4: Runtime Improvement from KV Caching

Method	**Tokens (sec)**	**Total Time (sec)**
Without KV Caching	157.613069	4.168436
With KV Caching	573.218355	1.454943

Speedup Analysis KV Caching provided a **3.64x** speedup. This is because the 'Without Cache' method re-computes the entire sequence at every step, while 'With Cache' only computes the single newest token.

Hence, KV caching significantly accelerates autoregressive decoding, especially for longer sequences, by avoiding repeated computation over the already-processed context.

6.3 Gradient Accumulation

To simulate training with larger effective batch sizes, we implement **gradient accumulation**. Instead of updating the model after every mini-batch, gradients are accumulated across multiple steps before performing a single optimization update.

We fix the mini-batch size at 32 and vary the number of accumulation steps:

- 1 steps $\Rightarrow$ effective batch size 32
- 2 steps $\Rightarrow$ effective batch size 64
- 4 steps $\Rightarrow$ effective batch size 128
- 8 steps $\Rightarrow$ effective batch size 256

Please note that to do this experiment it was not at all possible to run entire epochs for all the data (due to time and compute constraints)

Due to this, we have used the following:

- Data Subsize - 5000
- Number of epochs - 2000

Training Loss Comparison

Figure 8 shows training loss curves for all accumulation settings on a single plot.

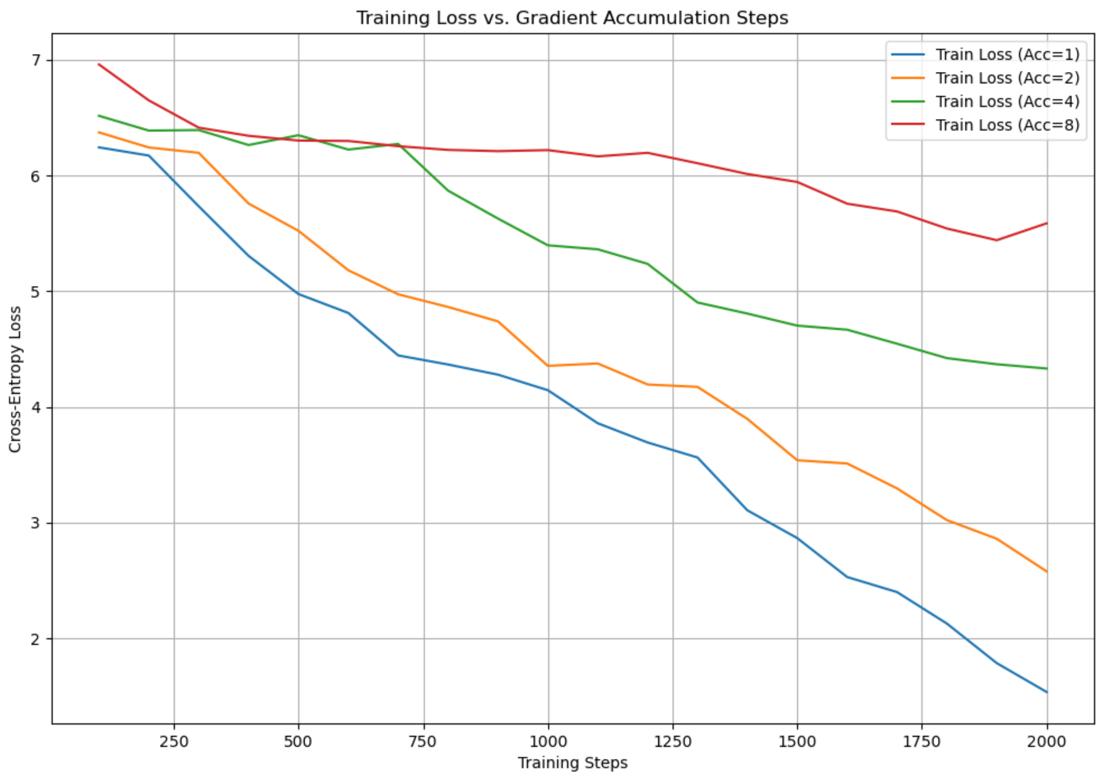

Figure 8: Training loss curves for different gradient accumulation settings.

The results from the plot are exactly as expected and clearly illustrate the fundamental trade-offs of gradient accumulation.

- **Stability and Smoothness** The most immediate observation is the effect on the stability of the loss curve.
  - **Train Loss (Acc=1):** The baseline (blue line) is the noisiest. The loss value fluctuates significantly, which is typical for small batch sizes (effective size 32), as individual "easy" or "hard" batches can heavily skew the gradient.
  - **Train Loss (Acc=8):** The (red) line is the smoothest. By averaging gradients over 8 steps before updating (effective size 256), the noise from individual batches is averaged out, leading to a much more stable and reliable descent.
- **Convergence Speed (Per Step)** The second observation is the convergence speed with respect to the number of training steps (the x-axis).
  - **Train Loss (Acc=1):** This curve (blue) shows the fastest decrease in loss per step and finishes at the lowest point. This is because the model's weights are updated at every single step. After 2,000 steps, it has performed 2,000 optimizer updates.

  - **Train Loss (Acc=8):** This curve (red) shows the slowest decrease in loss per step. This is not because the model is "worse," but because it is learning less frequently. After 2,000 steps, it has only performed 250 optimizer updates (2000 steps / 8).

Runtime Per Epoch

Table 5 summarizes the runtime per epoch for each configuration.

Table 5: Runtime per Epoch for Different Gradient Accumulation Settings

Accumulation Steps	**Effective Batch Size**	**Total Time for 2000 steps (sec)**
1	32	590.782046
2	64	618.625594
4	128	633.628232
8	256	683.342436

Following is the explanation for why the runtime is increasing slightly:

- **With Acc=1 (Baseline):** At every step, we do loss.backward() (calculate gradients) and optimizer.step() (apply gradients)

- **With Acc=8 (Accumulation):** At steps 1-7, we do loss.backward(). PyTorch not only calculates the new gradients but also adds them to the gradients it's already storing from the previous steps. At step 8, we do loss.backward() (calculate + add), and then optimizer.step() (apply all 8).

  That tiny "adding" operation at each step has a small computational cost. This small cost, repeated over thousands of steps, adds up to be slightly more than the time we save by calling optimizer.step() less often.

.