2. **Byte-Pair Encoding (BPE) Tokenizer**

   1. **Understanding Unicode**

      a. `chr(0)` returns '`b'\x00`"

      b. `__repr__()` shows an escaped, unambiguous form whereas its printed representation outputs the character itself which is actual NULL character(invisible).

      c. When we add it to text, it simply adds an invisible character at that position which may not be visually noticeable.

   2. **Unicode Encodings**

      a. UTF-8 is preferred over UTF-16 or UTF-32 due to its superior compression, universal applicability without out of vocabulary errors, and native compability with byte-stream data.

      b. If we give the input 牛, it breaks and gives an error "UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe7 in position 0: unexpected end of data". This is due to the fact that the decoding is done byte by byte and when the encoded character is multi-byte, it fails to decode it properly.

      c. `b'\xC3\x28`' is invalid in UTF-8 because 0xC3 indicates the start of a 2-byte character, but 0x28 is not a valid continuation byte.

   3. **BPE Training on TinyStories**

      a. It seems that the longest token is 'accomplishment' with a length of 15 bytes. The memory usage is around 158.3 MB and the training took approximately 61.72 seconds. It makes sense that longer tokens are formed from frequently occurring sequences of characters in the dataset, and 'accomplishment' might be a common word in the TinyStories dataset.

      b. Pre-tokenization part took the most time (40 seconds). Finding chunks and their word count seems to be the most time-consuming part of the process. Second highest is merging the tokens based on the pairs found (20 seconds).

   4. **Experiments with tokenizers**

      a. The tokenizer's compression ratio (bytes/token) is around 2.6673 in a sample of 10 documents from TinyStories dataset.

      b. The througput of the tokenizer is approximately 12862131.95 bytes/second. To tokenize the Pile dataset (about 825 GB), it would take around 178 hours (approximately 7.4 days).

      c. uint16 is appropriate choice for encoding the token IDs since the vocabulary size is 10,000 which fits well within the range of uint16 (0 to $2^{16} - 1 = 65535$).

3. **Transformer Language Model Architecture**

   1. **Transformer LM resource accounting**

      a. With the current implementation, GPT-2 XL has nearly 2,127,057,600 (≈2.13B) trainable parameters. At 4 bytes (32 bit floating point) per parameter, this amounts to approximately 8.5 GB of memory just for model parameters.

      b. Calculation is as follows:

         - **Per Transformer Block (L = 48 in total)**

- **Attention Projections (W_Q, W_K, W_V)**: 3 matrix multiplications,

$$= 3 * 2 * seq\_len * d\_model * d\_model$$
$$= 3 * (2 * 1024 * 1600 * 1600)$$
$$= \textbf{15,728,640,000 FLOPs}$$

- **Attention Scores (QK^T)**: 1 matrix multiplication,

$$= 2 * seq\_len^2 * num\_heads * (d\_model/num\_heads)$$
$$= 2 * 1024^2 * 25 * (1600/25)$$
$$= \textbf{3,355,443,200 FLOPs}$$

- **Attention Output($ScoresV$)**: 1 matrix multiplication,

$$= 2 * seq\_len^2 * num\_heads * (d\_model/num\_heads)$$
$$= 2 * 1024^2 * 25 * (1600/25)$$
$$= \textbf{3,355,443,200 FLOPs}$$

- **Attention Output Projection(W_O)**: 1 matrix multiplication,

$$= 2 * seq\_len * d\_model^2$$
$$= 2 * 1024 * 1600^2$$
$$= \textbf{5,242,880,000 FLOPs}$$

- **FFN(W_1, W_3)**: 2 matrix multiplications,

$$= 2 * (2 * seq\_len * d\_model * d\_ff)$$
$$= 2 * 2 * 1024 * 1600 * 6400$$
$$= \textbf{41,943,040,000 FLOPs}$$

- **FFN(W_2)**: 1 matrix multiplication,

$$= 2 * seq\_len * d\_ff * d\_model$$
$$= 2 * 1024 * 6400 * 1600$$
$$= \textbf{20,971,520,000 FLOPs}$$

So for all blocks combined, the total FLOPs is:

$$= 48 * (15,728,640,000 + 3,355,443,200 + 3,355,443,200 + 5,242,880,000$$
$$+ 41,943,040,000 + 20,971,520,000)$$
$$= 48 * 90,596,486,400$$
$$= \textbf{4,348,631,347,200 FLOPs}$$

- **Final LayerNorm and Output Projection**: 1 matrix multiplication,

$$= 2 * seq\_len * d\_model * vocab\_size$$
$$= 2 * 1024 * 1600 * 50257$$
$$= \textbf{164,682,137,600 FLOPs}$$

So the total FLOPs are:

$$= 4,348,631,347,200 \text{ (Blocks)} + 164,682,137,600 \text{ (Head)}$$
$$= \textbf{4,513,313,484,800 FLOPs}$$
$$\boxed{\approx \textbf{4.51 TFLOPs}}$$

c. Most of the FLOPs occur within the **Transformer blocks** (about 96.3% of total FLOPs) and within each block, the **FFN layers** contribute about 70% of the block's FLOPs (62.9 GFLOPs) compared to the attention mechanism (27.7 GFLOPs).

d. The breakdown of the FLOPs for each model is as follows:

- **GPT-2 Small (L = 12, d_model = 768, num_heads = 12)**
  - Blocks: $\mathbf{2.7 * 10^{11}}$ **FLOPs** (77.7% of total)
  - Final LayerNorm and Output Projection: $\mathbf{0.79 * 10^{11}}$ **FLOPs** (22.3% of total)
  - Total: Approximately **0.32 TFLOPs** per forward pass.
- **GPT-2 Medium (L = 24, d_model = 1024, num_heads = 16)**
  - Blocks: $\mathbf{9.27 * 10^{11}}$ **FLOPs** (90% of total)
  - Final LayerNorm and Output Projection: $\mathbf{1.05 * 10^{11}}$ **FLOPs** (10% of total)
  - Total: Approximately **1.03 TFLOPs** per forward pass.
- **GPT-2 Large (L = 36, d_model = 1280, num_heads = 20)**
  - Blocks: $\mathbf{2.1 * 10^{12}}$ **FLOPs** (94% of total)
  - Final LayerNorm and Output Projection: $\mathbf{1.32 * 10^{11}}$ **FLOPs** (6% of total)
  - Total: Approximately **2.23 TFLOPs** per forward pass.

As the model size increases, the proportional FLOPs cost changes toward the **Transformer blocks**, especially the FFN layers, which dominate the computational cost in larger models. The final output projection becomes relatively less expensive as the model size increases.

e. Increasing the context_length from 1024 to 16,384 (a 16x increase) results in a significant rise in total FLOPS, approximately **33.2x** (from 4.51 TFLOPs to around 149.8 TFLOPs). As the context length grows, the contribution to the FLOPS shifts from the FFN layers to the attention score calculation. This shift becomes more pronounced at longer context lengths, as **attention score calculations** become the dominant cost due to their quadratic scaling with sequence length. Consequently, attention score computations grow from **7%** to **55%** of total FLOPS, becoming the new bottleneck.

4. **Training a Transformer LM**

1. **Tuning the learning rate**

With lr = 1e1, the loss keeps decaying properly. Inital iterations see a sharp drop in loss, and then it continues to decrease steadily and slowly.

With lr = 1e2, the loss keeps fluctuating and diverging. It starts with a sharp drop but then it increaes and decreases erratically, indicating that the model is not converging properly.

With lr = 1e3, the loss diverges immediately and keeps increasing without any sign of convergence.

2. **Resource accounting for training with AdamW**

a. Let

$B = batch\_size$,

$T = context\_length$,

$D = d\_model$,

$H = num\_heads$,

$V = vocab\_size$,

$N = num\_layers$,

$d_{ff} = 4D$

- **Parameters**:
  - **Embeddgings**: Input ($\mathbf{V} \times \mathbf{D}$) + Position ($\mathbf{T} \times \mathbf{D}$)
  - **Transformer Blocks**: For each of the **N** blocks, we have:
    - Attention Projections (W_Q, W_K, W_V, W_O): **4** matrices of size ($\mathbf{D} \times \mathbf{D}$)
    - FFN (W_1, W_2, W_3): **3** matrices of size ($\mathbf{D} \times \mathbf{d_{ff}}$) and ($\mathbf{d_{ff}} \times \mathbf{D}$)
    - **LayerNorms**: **2** RMSNorms of size ($\mathbf{D}$)
  - **Final Layers**: Final RMSNorm ($\mathbf{D}$) and Output Projection ($\mathbf{D} \times \mathbf{V}$)

  So the total memory for parameters ($\mathbf{N_{params}}$) is:

$$\mathbf{N_{params}} = (\mathbf{V} \times \mathbf{D}) + (\mathbf{T} \times \mathbf{D}) + \mathbf{N} * [\mathbf{4} * (\mathbf{D^2}) + \mathbf{3} * (\mathbf{D} \times \mathbf{d_{ff}}) + \mathbf{2} * \mathbf{D}] + (\mathbf{D} + \mathbf{D} \times \mathbf{V})$$

$$\boxed{\mathbf{N_{params}} = \mathbf{2}(\mathbf{VD}) + (\mathbf{TD}) + \mathbf{N} * [\mathbf{16}(\mathbf{D^2}) + \mathbf{2}(\mathbf{D})] + (\mathbf{D})}$$

$$\boxed{\mathbf{Memory_{params}} = \mathbf{N_{params}} * \mathbf{4\ bytes}}$$

- **Gradients**: We store one gradient value for every parameter, so

$$\boxed{\mathbf{Memory_{grads}} = \mathbf{N_{params}} * \mathbf{4\ bytes}}$$

- **Optimizer State**: AdamW maintains two additional values (momentum and variance) for each parameter, so

$$\boxed{\mathbf{Memory_{optimizer}} = \mathbf{N_{params}} * \mathbf{2} * \mathbf{4\ bytes}}$$

- **Activations**:
  - **Transformer Block**:
    - RMSNorm1 Input: $\mathbf{B} \times \mathbf{T} \times \mathbf{D}$
    - QKV Projections (Input/Output): $\mathbf{B} \times \mathbf{T} \times \mathbf{D}$ (Input) + $\mathbf{3} \times \mathbf{B} \times \mathbf{T} \times \mathbf{D}$ (Output)
    - $Q^T K$ Scores & Softmax (Probs): $\mathbf{2} \times \mathbf{B} \times \mathbf{H} \times \mathbf{T} \times \mathbf{T}$
    - Weighted Sum (Context): $\mathbf{B} \times \mathbf{T} \times \mathbf{D}$
    - Attention Output Projection: $\mathbf{B} \times \mathbf{T} \times \mathbf{D}$
    - RMSNorm2 Input: $\mathbf{B} \times \mathbf{T} \times \mathbf{D}$
    - FFN W1 Output: $\mathbf{B} \times \mathbf{T} \times \mathbf{d_{ff}}$
    - FFN W2(SiLU) Output: $\mathbf{B} \times \mathbf{d_{ff}} \times \mathbf{T}$
    - FFN W3 Output: $\mathbf{B} \times \mathbf{T} \times \mathbf{d_{ff}}$

In total, the activations for one block are $\mathbf{20(BTD) + 2(BHT^2)}$.

– **Final Layers**: Final Norm Input: $\mathbf{B \times T \times D}$, Output Projection (logits): $\mathbf{B \times T \times V}$ So the total memory for activations ($\mathbf{N_{activations}}$) is:

$$\mathbf{N_{activations} = N * [20(BTD) + 2(BHT^2)] + BTD + BTV}$$

$$\boxed{\mathbf{Memory_{activations} = N_{activations} * 4 \text{ bytes}}}$$

Total Peak Memory:

$$\mathbf{Memory_{total} = Memory_{params} + Memory_{grads} + Memory_{optimizer} + Memory_{activations}}$$

$$\mathbf{Memory_{total} = N_{params} * 4 + N_{params} * 4 + N_{params} * 8 + N_{activations} * 4}$$

$$\boxed{\mathbf{Memory_{total} = 16 * N_{params} + 4 * N_{activations} \text{ bytes}}}$$

b. For GPT-2 XL model, we have
$B = batch\_size$,
$T = 1024$,
$D = 1600$,
$H = 25$,
$V = 50257$,
$N = 48$,
$d_{ff} = 6400$

- **Parameters**: $\mathbf{N_{params}} = 2(VD) + (TD) + N * [16(D^2) + 2(D)] + D \approx 2.13B$ parameters, which amounts to approximately **8.5 GB** of memory (4 bytes/parameter).
- **Gradients**: Same as parameters, so approximately **8.5 GB** of memory.
- **Optimizer State**: 2 additional values per parameter, so approximately **17 GB** of memory.
- **Activations**: $\mathbf{N_{activations}} = N * [20(BTD) + 2(BHT^2)] + BTD + BTV$
  Taking B out of the equation, we have
  $\mathbf{N_{activations}} = B * [N * (20TD + 2HT^2) + TD + TV]$
  For GPT-2 XL, this is approximately $\mathbf{B * 4.1B}$ activations, which amounts to approximately **16.4 GB * B** of memory.

So the required expression:

$$\boxed{\mathbf{Memory_{total}(GB) = 16.4 \cdot B + 34}}$$

For this to fit in 80 GB memory, we need batch_size $\mathbf{B \leq 2.8}$, so the maximum batch size we can use is **2**.

c.

$$\boxed{\mathbf{FLOPs_{AdamW} \approx 15 \times N_{params}}}$$

Justification: AdamW performs element-wise operations on the parameters. For each parameter, the update involves:

- Updating first moment m: **3 operations** (mul, mul, add).

- Updating second moment v: **4 operations** (mul, mul, mul, add).
- Parameter update θ: **8 operations** (sqrt, add, div, mul, sub, mul, mul, sub).

Total is around 15 FLOPs per parameter.

d.
- **Total Training FLOPs**: $400k \times B \times 6 \times N_{params}$. Here 6 comes from 2 forward passes and 4 for backward passes.
  Total FLOPs $= 400k \times 1024 \times 6 \times 2.13B \approx \mathbf{5.2 \times 10^{18}}$ FLOPs.
- **Hardware Throughput**: Peak throughput = 19.5 teraFLOPs/s. With 50% MFU, effective throughput = 9.75 teraFLOPs/s.
- **Total Training Time**: $\frac{5.2\times10^{18} \text{ FLOPs}}{9.75\times10^{12} \text{ FLOPs/s}} \approx \mathbf{533,333}$ seconds $\approx \mathbf{148.15}$ hours $\approx \boxed{\mathbf{6.17} \text{ days}}$.

5. **Training loop**

- The training loop is implemented in the script located at `student/train.py`.

6. **Generating text**

- The decoding/generation script lives at `student/generate.py`.

7. **Experiments**

1. **Tune the learning rate**

a. I employed the stratery to sweep through the learning rates geometrically (1e-5, 1e-4, 1e-3, 1e-2, 1e-1) and observed the loss curves to identify the optimal learning rate for training. The learning rates are given in the following plot:

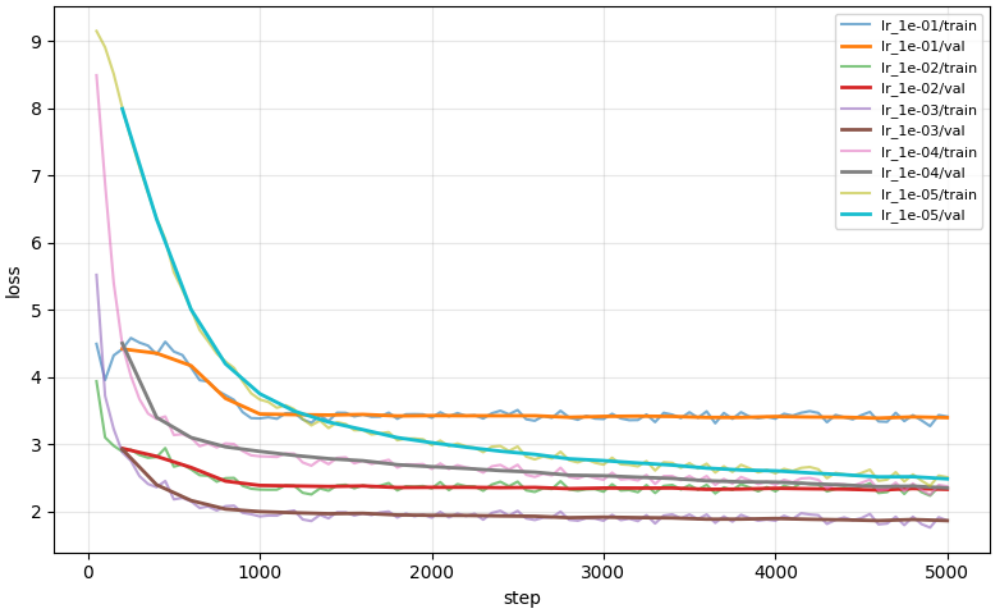

Figure: Loss curves for different learning rates.

I ran the training with batch size of 256, the optmial learning rate of 1e-3 as seen from above experiments and with cosine learning rate decay schedule tip given in the document and I was able to get a model with final loss of around 1.3. The learning curve is as follows:

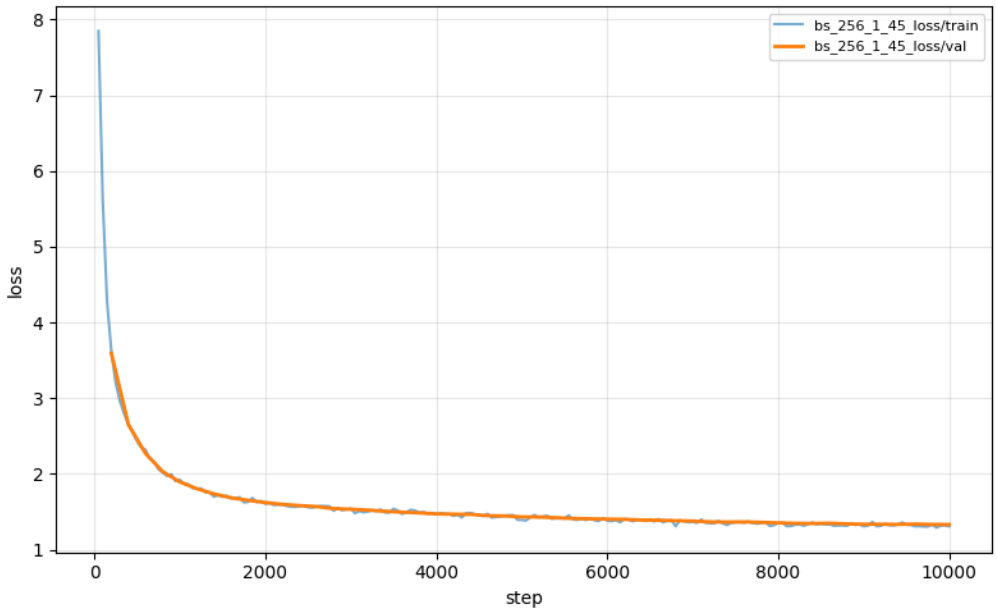

Figure: Loss curve for optimal learning rate of 1e-3.

b. Based on the loss curves below, the optimal learning rate for training the model is **1e-3**, as it shows a steady decrease in loss without divergence or erratic behavior, unlike higher learning rates, especially 1e-1, which diverges from the start itself. Having higher learning rate (1e-1) results in divergence because the model takes too large steps in parameter space, overshooting the optimal minima and causing the loss to not decrease properly.

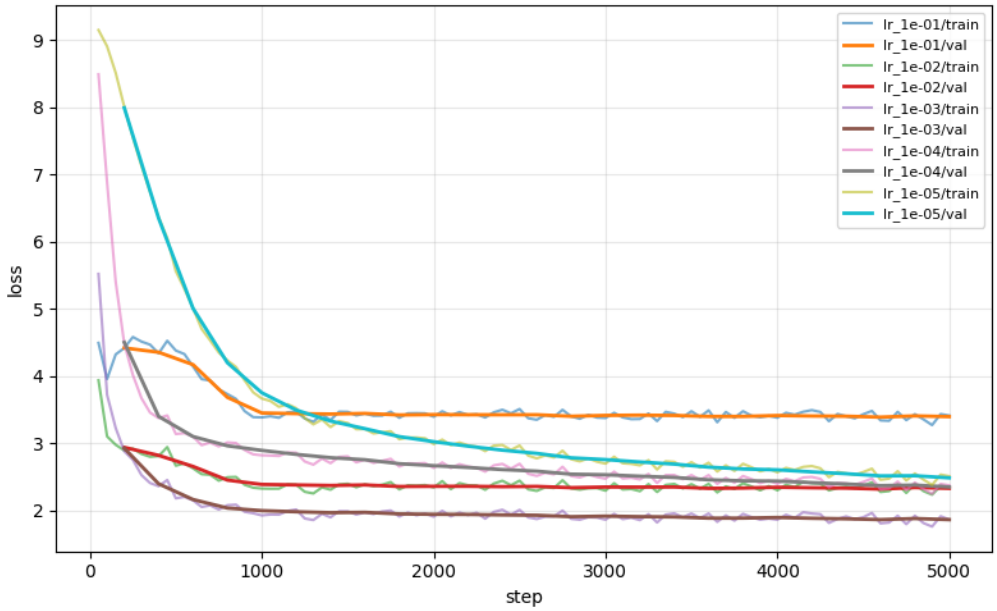

Figure: Loss curves with at least one divergent learning rate.

2. **Batch size variations** The learning curves for different batch sizes are given below:

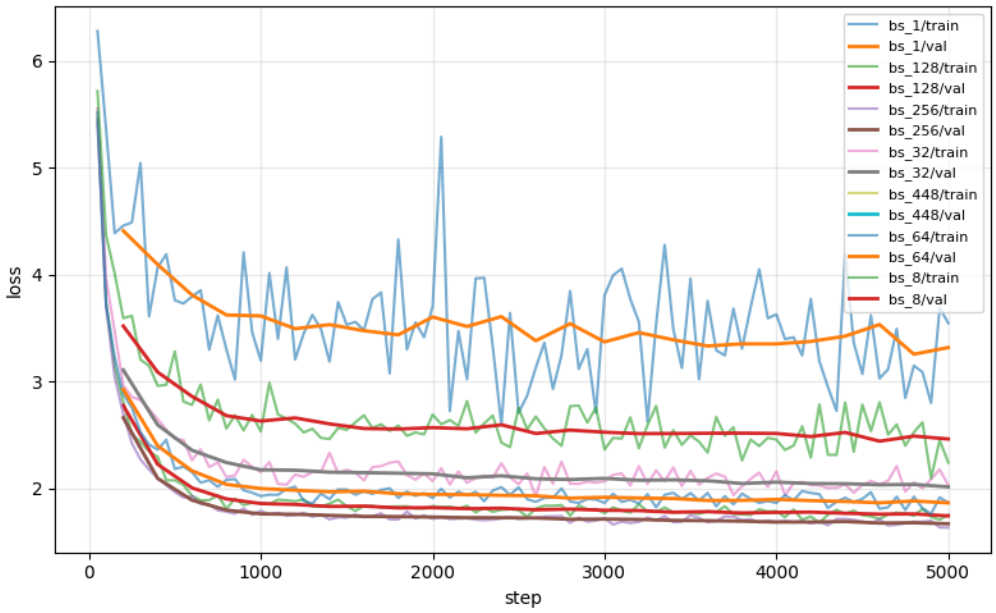

Figure: Loss curves for different batch sizes.

Batch size of 1 and 8 seemed erratic in training losses. But generally, as the batch sizes increase, the losses decrease more for the iterations trained. But at the same time, the training time per iteration also increases significantly. So there is a trade-off between faster convergence (in terms of iterations) and longer training time per iteration. Here, a batch size of 448 was the GPU memory limit so 256 batch size had the best performance in terms of loss reduction while still being able to train within a reasonable time frame. Having higer batch sizes results in better loss reduction because it provides a more accurate estimate of the gradient, leading to more stable and effective updates. However, it also increases the training time per iteration due to the larger amount of data being processed in each step.

3. **Generate text**

   With the prompt as "Once upon a time, Siva", the text generated by 256 batch size model is as follows:

   Once upon a time, Siva was a little girl who loved to play with her friends. One day, she found a big, red ball in the park. She was so happy! Bet asked her friends to play with the ball. They all said yes, and they all played with the ball. They had so much fun. But then, something unexpected happened. The ball went up into a tree! Pig, Bop, and her friends looked up and saw a big, red ball stuck in the tree. They all wanted to get the ball down. So, they all tried to jump and get the ball down. But the ball was too high up, and she could not reach it. Chew and her friends were sad. They could not get the ball back. They went home without the ball. In the end, they all lost the ball and could not get the ball back. </endoftext/>

   The output is very fluent. The model is able to generate coherent sentences and maintain a consistent narrative flow. Lower loss is generally one factor that correlates with better generation quality. Another factor is temperature provided while decoding the text. Changing temperature values, I noticed that higher temperature (e.g., 1.2) produced more diverse and creative outputs, while lower temperature (e.g., 0.5) resulted in more conservative and repetitive text.

4. **Remove RMSNorm and train**

Learning curve when rms norm is removed and trained at the previously found optimal learning rate of 1e-3 is given below:

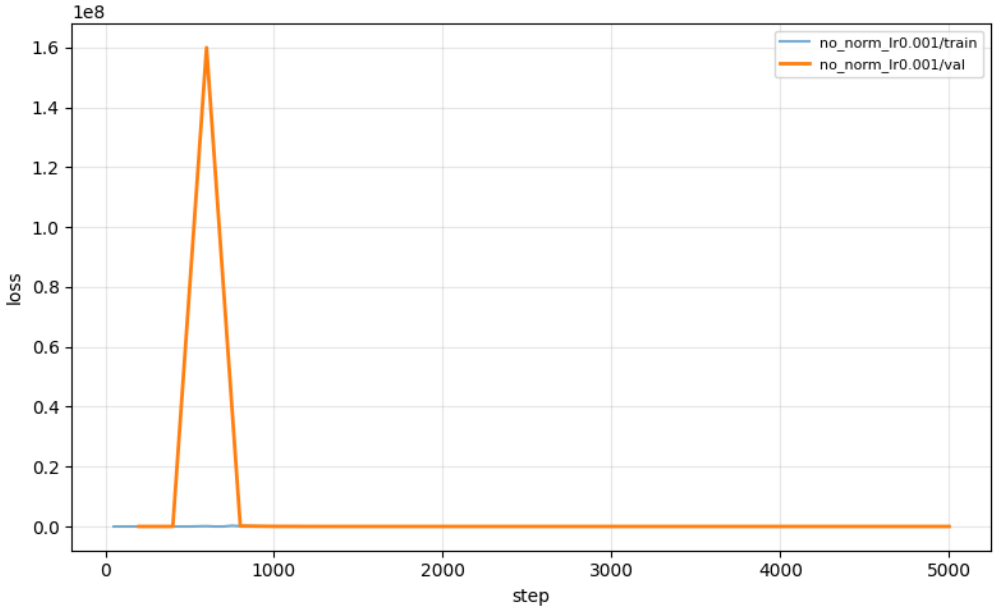

Figure: Loss curve without RMSNorm at optimal learning rate of 1e-3.

The loss curve without RMSNorm shows a much more erratic behavior compared to the original model with RMSNorm. The loss fluctuates significantly at the start indicating that the model is struggling to learn effectively without normalization. This highlights the importance of RMSNorm in stabilizing training and facilitating better convergence. The final loss after 5k iterations is around 2.4 which is much higher than the original model's final loss of around 2.0, indicating that the model without RMSNorm is not learning as effectively as the original model.

Running at lower learning rates (1e-4 and 1e-5) helped in steady decrease in loss, but the final loss after 5k iterations was still higher than the original model with RMSNorm, further confirming that RMSNorm plays a crucial role in the training process and helps the model converge to a better solution. The following plot shows the learning curves at different learning rates without RMSNorm (bs_32 is the base model with rms norm and lr 1e-3 for reference):

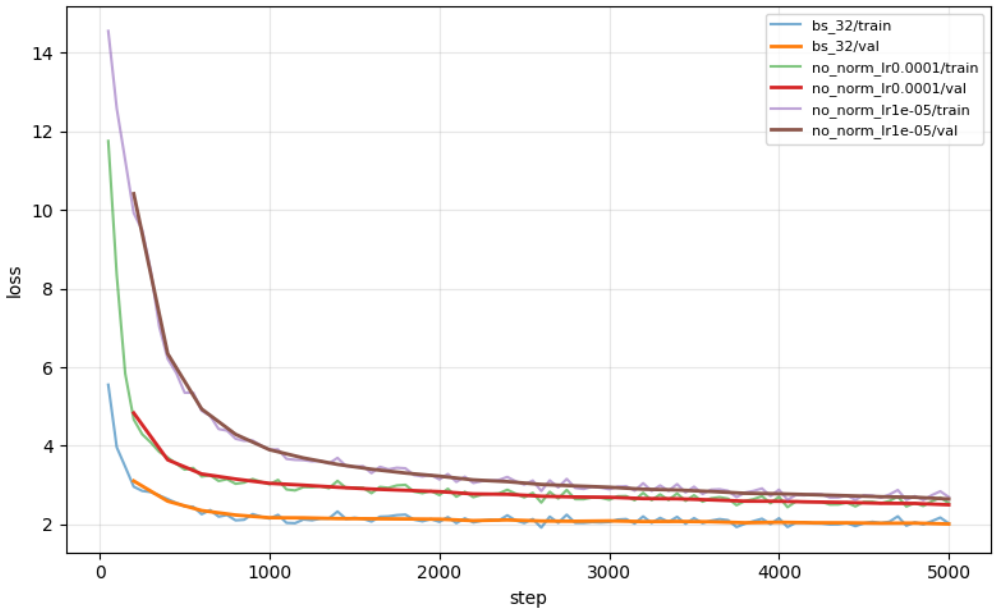

Figure: Loss curves without RMSNorm at lower learning rates.

5. **Implement post-norm and train**

   The learning curve for the model with post-norm vs our baseline model is given below:

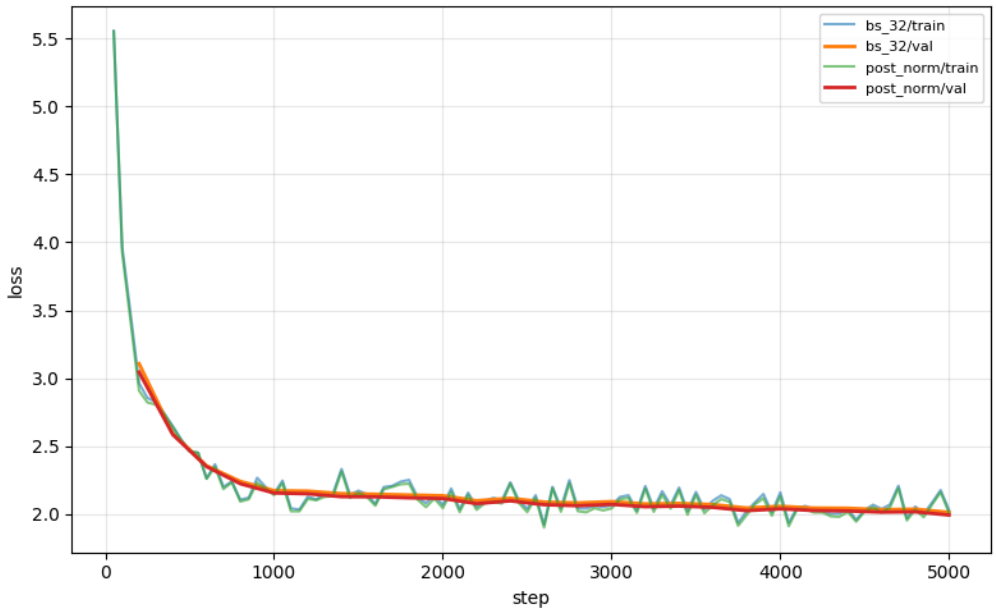

Figure: Loss curve with post-norm at optimal learning rate of 1e-3.

   Both the models perform very similarly in terms of loss reduction. The post norm one seems to have a very slightly lesser loss at the end of 5k iterations and also in the val loss, but the difference is not that significant.

6. **Implement NoPE**

   The learning curve for the model with NoPE vs our baseline model is given below:

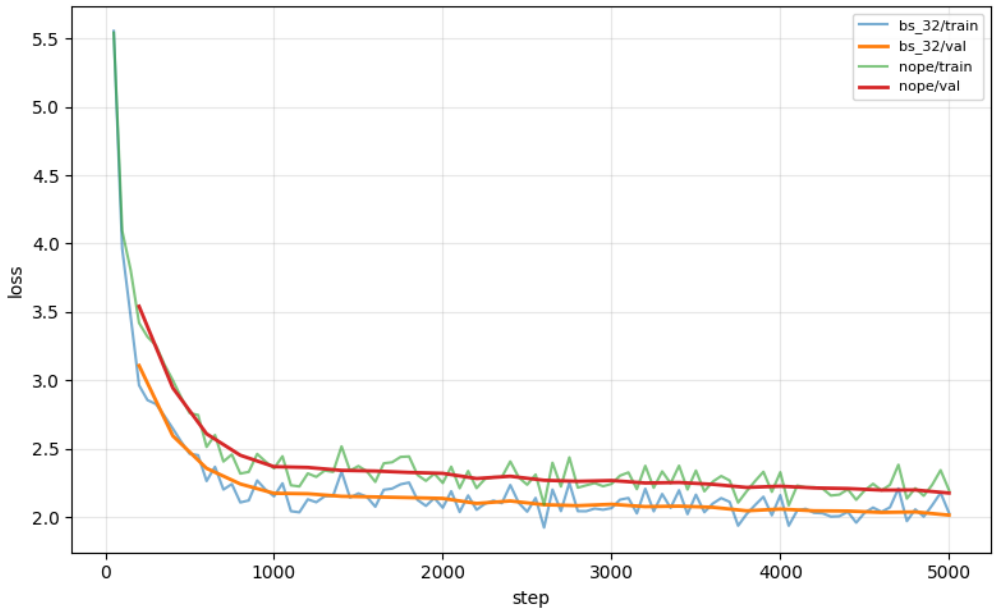

Figure: Loss curve with NoPE at optimal learning rate of 1e-3.

The model with NoPE performs significantly worse than the baseline model with positional embeddings. The loss curve for the NoPE model shows a much higher loss throughout the training process, indicating that the model is struggling to learn effectively without positional information. This highlights the importance of positional embeddings in helping the model understand the order of tokens in the sequence, which is crucial for language modeling tasks.

7. **SwiGLU vs SiLU** The learning curve for the model with SwiGLU vs our baseline model is given below:

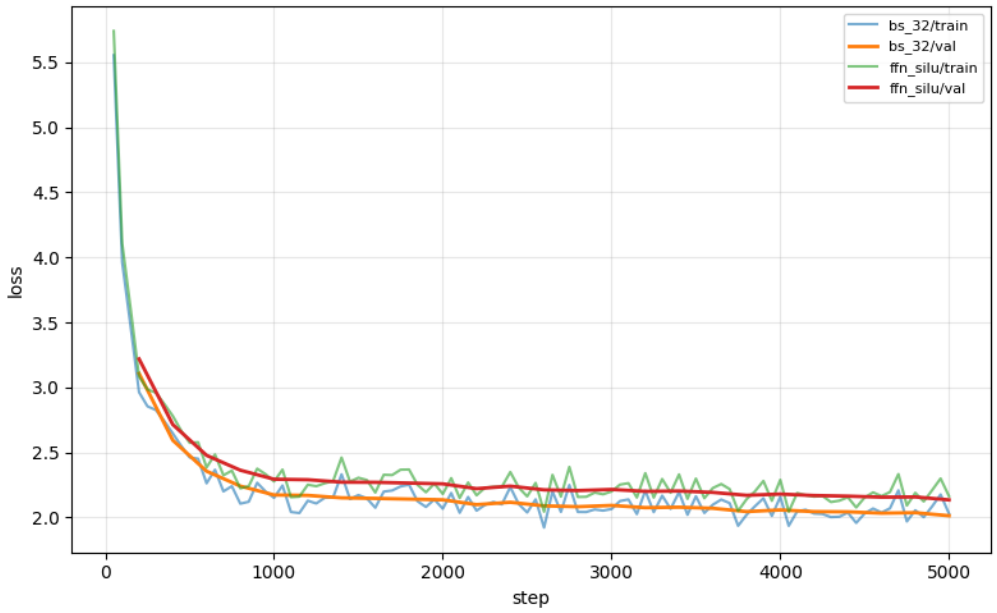

Figure: Loss curve with SwiGLU at optimal learning rate of 1e-3.

The model with SwiGLU performs better than the baseline model with SiLU activation. The loss for SwiGLU is consistently lower than the SiLU model throughout the training process, indicating that

SwiGLU is more effective in capturing complex patterns in the data and facilitating better learning compared to SiLU. This suggests that using SwiGLU as the activation function in the FFN layers can lead to improved performance in transformer language models.