Assignment 1

Vincent Chan
University of Hawaii
ECE491B: Introduction to Large-Scale AI Systems
vchan26@hawaii.edu

February 17, 2025

1 Introduction

In this paper, we present the implementation of a transformer-based language model from scratch, leveraging a minimal set of PyTorch methods. The goal of this project was to gain a deeper understanding of the transformer architecture by building it from the ground up, following the guidelines outlined in the assignment handout and conducting experiments. After completing this project, I gained valuable insights into the inner workings of transformer models, especially on how each component works and interacts with each other to transform an input sequence into an output distribution that we can sample to generate text.

All implementations and code are available on my GitHub Repository (Appendix A). The experimentation and model training were conducted on KOA, the High-Performance Computing (HPC) cluster at the University of Hawaii at Manoa. I performed most of the training tasks using a NVIDIA Nvidia RTX-5000 GPU with 16 GB of VRAM. Additionally, just to test it out, I was able to utilize a NVIDIA H100 GPU to train one model, allowing for a comparison of performance with more tokens processed.

This paper details the design, implementation, and evaluation of a basic transformer model, along with the results of several experiments to assess its effectiveness and performance.

2 Byte-Pair Encoding (BPE) Tokenizer

2.1 Understanding Unicode

What Unicode character does `chr(0)` return?

`\x00` which is `b'\x00'` in bytes. In Unicode, this represents the `NULL` character.

How does this character's string representation (`__repr__()`) differ from its printed representation?

This character's string representation using `__repr__()` results in the string '`\x00`'; however, the character's printed representation results in nothing.

What happens when this character occurs in text?

When the string is evaluated, it appears as `\x00`. When the string is printed, it prints nothing, leaving no character there.

2.2 BPE Tokenizer Training

The BPE Tokenizer trainer is implemented in train_bpe.py. The general process involves identifying the frequency of all token pairs, merging the most frequent pair, and repeating this process until the

vocabulary limit is reached.

This high-level approach is inefficient and results in long training times. To optimize this process, I implemented the following improvements, which can be viewed in the linked GitHub repository. First, I pretokenize the text using the GPT-2 pretokenizer regex and construct a frequency table of all pretokens. This allows me to adjust the frequency of a pair based on the frequency of the pretoken in which it appears. Another optimization is to maintain a mapping of pretokens containing each pair, so that only affected pretokens are updated during merges, rather than recomputing for the entire list.

With these optimizations, my BPE implementation completes the pytest (test_train_bpe.py) run in approximately 0.4 seconds, and trains on the TinyStories dataset in around 18 minutes. Training on OpenWebText completes in under five hours.

2.3 BPE Experiments

When training BPE on the TinyStories dataset, the most time-consuming step is pretokenization. Following that, finding the most frequent pair takes the next longest time. According to psutil, the peak memory usage was 132.89 MB. The breakdown of execution time is as follows:

- Pre-tokenization: 1039.8350 sec
- Constructing word list: 0.0446 sec
- Computing pair frequencies: 0.2632 sec
- Merging pairs: 53.1487 sec
  - Total time spent finding max pair across all iterations: 49.3114 sec
  - Total time spent processing affected occurrences across all iterations: 3.0866 sec
- **Total execution time: 1093.2916 sec**

The longest token in the vocabulary is Longest token: b' accomplishment' (Length: 15). This makes sense since we are using BPE which would break words up into subwords. "accomplish" and "ment" are probably two common subwords and they would get merged together.

For OpenWebText, pretokenization takes proportionally less time compared to the merging process, this could be due to the higher vocabulary size requiring much more searching for max pairs and updating the affected occurrences since each pair may be present in even more words.

The longest token in from training the BPE on the OWT is ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ. While it seems like complete nonsense it actually exists quite often if you browse through the dataset. This is likely the result of some errors in the data collection and cleaning process.

2.4 BPE tokenizer

The tokenizer initially splits the text to encode on special tokens. This prevents a chunk from cutting off a special token. A drawback of my approach is that in some cases if the text goes on very long without a special token there may be some memory concerns.

2.5 BPE Tokenizer Experiments

(a) The TinyStories tokenizer achieves a compression ratio of 4.08 bytes/token, while the OpenWebText tokenizer achieves 4.50 bytes/token, indicating that the latter is slightly more efficient at encoding OpenWebText samples.

(b) When tokenizing OpenWebText with the TinyStories tokenizer, the compression ratio drops to 3.40 bytes/token, suggesting that the TinyStories tokenizer is less efficient for OpenWebText, likely due to its smaller vocabulary, leading to longer token sequences.

(c) The estimated throughput is 178.8 KB/s for TinyStories and 478.7 KB/s for OpenWebText. At these speeds, tokenizing The Pile (825GB) would take around 1.34 hours using the OpenWebText

tokenizer and around 3.97 hours using the TinyStories tokenizer.

(d) Storing token IDs as uint16 is appropriate because both vocabularies are $\leq$ 65,536 tokens, meaning each token fits within 2 bytes, reducing memory usage compared to uint32 (4 bytes/token) while avoiding truncation and uint8 is way too small to store a vocabulary of 32,000

3 Transformer language model architecture

The the transformer is implemented in parts and follows the diagram of the transformer model very closely. All the code for the transformer model can be found at transformer_model.py and is written from scratch; barring a few allowed PyTorch definitions.

3.1 Transformer resource accounting

3.1.1 Model parameter accounting

Consider GPT-2 XL, which has the following configuration. We can use this python script to calculate the total trainable parameters.[1]

The trainable parameters are contributed by:

- Token embeddings: $V \times d_{\text{model}}$
- Position embeddings: $C \times d_{\text{model}}$
- **Transformer Blocks:** For each of the L layers, we have:
  - RMSNorms: $2\,d_{\text{model}}$,
  - Multi-head self-attention (MHSA): 4 projection matrices of size $d \times d$, i.e. $4\,d^2$,
  - Feed-forward network (FFN): Two matrices: $d \times d_{ff}$ and $d_{ff} \times d$, totaling $2\,d \times (4d) = 8\,d^2$.

  Thus, per block:
  $$2d + 4d^2 + 8d^2 = 2d + 12d^2.$$
- Final RMSNorm: d_{model}.
- LM head: A projection from d_{model} to V, i.e. $d \times V$.

Therefore, the total number of trainable parameters is:

$$d_{\text{model}} \times C + d_{\text{model}} \times V + (12d_{\text{model}}^2 + 2d_{\text{model}}) \times L + d_{\text{model}} \times V + d_{\text{model}} \tag{1}$$

```
vocab_size = 50257
context_length = 1024
num_layers = 48
d_model = 1600
num_heads = 25
d_ff = 6400

norm_weight = d_model
token_embedding_weights = vocab_size*d_model
position_embedding_weights = context_length*d_model
mhsa_weight = d_model*d_model * 4
ffn = d_model * d_ff + d_ff * d_model
block_weights = mhsa_weight + norm_weight + norm_weight + ffn
layer_weights = num_layers * block_weights
lm_head_weights = d_model * vocab_size
```

[1] Note: the num_heads do not contribute to the number of parameters since each head in the multi head self attention mechanism works on space d_model / num_heads such that the Q,K,V projections still take up $d\_model \times d\_model$.

```
trainable_parameters = token_embedding_weights + position_embedding_weights + layer_weights
    + norm_weight + lm_head_weights

>>> trainable_parameters
1637176000
>>> final_memory_in_bytes = trainable_parameters * 4
6548704000
```

We see that this model will take 6548704000 bytes, or approximately 6.55 gigabytes, to even load.

3.1.2 Model FLOPs

Now lets calculate the amount of FLOPs a single forward pass of the transformer does given C context length and d_{model}. A single matrix multiply of $(m \times n)$ and $(n \times p)$ needs $2mnp$ FLOPs to compute.

3.1.2.1 MHSA FLOPs

Each Transformer block includes a multi-head self-attention mechanism, which consists of several matrix multiplications.

Computing Query, Key, and Value Matrices

Each input token X (of shape $C \times d_{\text{model}}$) is projected into Query, Key, and Value matrices using learned weight matrices:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Each weight matrix W_Q, W_K, W_V has dimensions $d_{\text{model}} \times d_{\text{model}}$. Computing each of these requires:

$$C \times d_{\text{model}} \times d_{\text{model}}$$

Since we perform this for all three matrices (Q, K, V), the total FLOPs are:

$$\text{FLOPs}_{\text{QKV}} = 3 \times 2 \times C \times d_{\text{model}}^2 \tag{2}$$

Computing Attention Scores and Output

The attention mechanism is defined as:

$$\textit{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

The matrix multiplications in this computation are $QK^\top$ and $\text{softmax}(X)V$, where:

- Q has dimensions $C \times d_{\text{model}}$
- $K^\top$ has dimensions $d_{\text{model}} \times C$
- The softmax output has dimensions $C \times C$
- V has dimensions $C \times d_{\text{model}}$

Thus, the total FLOPs for the attention mechanism and output projection are:

$$\text{FLOPs}_{\text{Attention}} = 2C^2 d_{\text{model}} + 2C^2 d_{\text{model}} = 4C^2 d_{\text{model}}. \tag{3}$$

Final Projection

The attention output from the multi-head self-attention mechanism is projected back into the model dimension using the following operation:

$$\text{MultiHeadAttn} = \text{Concat}(head_1, \ldots, head_n)W_O$$

where W_O is a weight matrix of size $d_{\text{model}} \times d_{\text{model}}$, and the concatenated heads have a dimension of $C \times d_{\text{model}} \times n$, where n is the number of heads.

The matrix multiplication of the concatenated attention heads with W_O requires:

$$\text{FLOPs}_{\text{output projection}} = 2 \times C \times d_{\text{model}}^2 \tag{4}$$

Total MHSA FLOPs per Layer

Summing up all components:

$$\text{FLOPs}_{\text{MHSA}} = 8 \times C \times d_{\text{model}}^2 + 4 \times C^2 \times d_{\text{model}} \tag{5}$$

3.1.2.2 Feedforward Network (FFN) FLOPs

Each Transformer block also contains a Position-Wise Feed-Forward Network:

$$FFN(x) = \text{GELU}(xW_1)W_2$$

where W_1 has dimensions $d_{\text{model}} \times d_{\text{ff}}$ and W_2 has dimensions $d_{\text{ff}} \times d_{\text{model}}$

The FLOPs required for each transformation are:

$$\text{FLOPs}_{\text{expand}} = 2 \times C \times d_{\text{model}} \times d_{\text{ff}} \tag{6}$$

$$\text{FLOPs}_{\text{contract}} = 2 \times C \times d_{\text{ff}} \times d_{\text{model}} \tag{7}$$

Summing them gives the total FFN FLOPs per layer:

$$\text{FLOPs}_{\text{FFN}} = 2 \times C \times d_{\text{model}} \times d_{\text{ff}} + 2 \times C \times d_{\text{ff}} \times d_{\text{model}} = 4 \times C \times d_{\text{model}} \times d_{\text{ff}} \tag{8}$$

Note: Since d_{ff} is normally $4 \times d_{\text{model}}$, the expression can simplify to:

$$\text{FLOPs}_{\text{FFN}} = 16 \times C \times d_{\text{model}}^2 \tag{9}$$

Final Language Modeling Head

The final model output maps to the vocabulary space:

$$\text{FLOPs}_{\text{LM Head}} = 2 \times C \times d_{\text{model}} \times V \tag{10}$$

Total FLOPs for a Forward Pass

Summing the contributions from all L transformer layers and the LM head:

$$\text{FLOPs}_{\text{total}} = L \times (\text{FLOPs}_{\text{MHSA}} + \text{FLOPs}_{\text{FFN}}) + \text{FLOPs}_{\text{LM Head}} \tag{11}$$

Component	FLOPs per Layer
QKV Projection	$6Cd_{\text{model}}^2$
Attention Score Computation	$4C^2d_{\text{model}}$
Output Projection	$2Cd_{\text{model}}^2$
Total MHSA	$8Cd_{\text{model}}^2 + 4C^2d_{\text{model}}$
FFN Expansion (W_1)	$2Cd_{\text{model}}d_{\text{ff}}$
FFN Contraction (W_2)	$2Cd_{\text{ff}}d_{\text{model}}$
Total FFN	$16Cd_{\text{model}}^2$
Total per Layer	$8Cd_{\text{model}}^2 + 4C^2d_{\text{model}} + 16Cd_{\text{model}}^2$
Total for All Layers (L)	$L\times$ (Total per Layer)
LM Head Computation	$2Cd_{\text{model}}V$
Final FLOPs Estimate	$2Cd_{\text{model}}V + L\times$ (Total per Layer)

Table 1: FLOP Breakdown for a Forward Pass in GPT-2 XL

Component (FLOPs)	GPT-2 Small	GPT-2 Medium	GPT-2 Large	GPT-2 XL
MHSA Projections	57.98B **(13.25%)**	206.16B **(12.46%)**	483.18B **(10.89%)**	1006.63B **(9.19%)**
MHSA Attention	38.65B (19.88%)	103.08B (24.93%)	193.27B (27.23%)	322.12B (28.71%)
FFN	115.96B (39.76%)	412.32B (49.86%)	966.37B (54.46%)	2013.27B (57.41%)
LM Head	79.05B (27.10%)	105.40B (12.75%)	131.75B (7.42%)	164.68B (4.70%)
Total FLOPs	0.29T	0.83T	1.77T	3.51T

Table 2: FLOPs required for different GPT-2 model with context length 1024 and vocab 50257.

3.2 FLOPs analysis

We can use the derivation above to calculate an estimate of the number of FLOPs to do a single forward pass of various models. The calculations assume a context length of 1024 and vocabulary size of 50257.

As shown in Table 2, the smaller the model, the more evenly distributed the FLOPs are across the components. In the smallest model, we observe that MHSA, FFN, and LM Head each contribute roughly a third of the total FLOPs. However, as the model scales with more layers and a larger d_{model}, the FFN component starts to dominate in computational cost.

For the multi-head self-attention (MHSA) component, we can see that the projections require significantly more computation than the attention mechanism which makes sense since the matrix multiplications of the projections scale quadratically with d_{model} whereas the attention mechanism scales linearly. Similarly, the FFN component also scales quadratically with with d_{model} but has more even more matrix multiplications than the projections.

The results of increasing the context size to 16,384 are shown in Table 3. We can see that the attention mechanism becomes the dominant computational bottleneck, consuming the majority of the total FLOPs. This shift occurs because the self-attention mechanism scales quadratically with context length, whereas other components (projections, FFN, and LM Head) scale only linearly. Consequently, at large context sizes, attention computation far outweighs the cost of other operations.

4 Training a Transformer LM

4.1 Learning rates

I ran sgd.py for learning rates of [1e-3, 1e1, 1e2, 1e3]. With 1e-3 the loss barely changed at all. Increasing the learning rate to 1e1 lowered the loss value to around 3.9. Continuing to increase the learning rate with 1e2 resulted in an even lower loss of 3.2e-23. However, with a learning rate of 1e3

Component	GPT-2 XL (Ctx = 1024)	GPT-2 XL (Ctx = 16,384)
MHSA Projections FLOPs	1006.63B	16106.13B
MHSA Attention FLOPs	322.12B	82463.37B
FFN FLOPs	2013.27B	32212.25B
LM Head FLOPs	164.68B	2634.91B
Total FLOPs	3.51T	133.42T
MHSA Projections (%)	28.71%	12.07%
MHSA Attention (%)	9.19%	61.81%
FFN (%)	57.41%	24.14%
LM Head (%)	4.70%	1.97%

Table 3: FLOPs required for GPT-2 XL model with different context sizes (1024 vs. 16,384).

the loss diverges rapidly and ends with a final value of 1.8e18. This suggests that we need to carefully find a balance between too low and too high.

4.2 AdamW resource accounting

All the code for optimization and calculating loss is in optimizing.py

To account for the resources needed for the AdamW optimizer, we need to take account the following: model parameters, activations, gradients, and the optimizer state itself.
For the equations let us define the following: B batch size, C context length, V vocabulary size

4.2.1 Parameters

We can reference Equation 1 for the total parameters of the model which is given as:

$$d_{\text{model}} \times C + d_{\text{model}} \times V + (12d_{\text{model}}^2 + 2d_{\text{model}}) \times L + d_{\text{model}} \times V + d_{\text{model}}$$

4.2.2 Activations

For each forward pass, we need to store the maximum activation size for the following components:

- **Within each Transformer block (per block):**
  - **RMSNorm(s):** $2 \times (B \times C \times d_{\text{model}})$,
  - **Multi-head Self-Attention (MHSA) sublayer:**
    - QKV projections: $3 \times (B \times C \times d_{\text{model}})$,
    - QK^T matrix multiply: $B \times C^2$,
    - Softmax: $B \times C^2$,
    - Weighted sum and output projection: $2 \times (B \times C \times d_{\text{model}})$.

    The total memory for the MHSA sublayer is:

$$5BCd_{\text{model}} + 2BC^2. \tag{12}$$

  - **Position-wise Feed-forward Network (FFN):**
    - First linear (W_1): $B \times C \times d_{\text{ff}}$,
    - GELU activation: $B \times C \times d_{\text{ff}}$,
    - Second linear (W_2): $B \times C \times d_{\text{model}}$.

    The total memory for the FFN is:

$$9BCd_{\text{model}} \quad (\text{since } d_{\text{ff}} = 4d_{\text{model}}). \tag{13}$$

Hence, per Transformer block, the total activations are:

$$2BCd_{\text{model}} + (5BCd_{\text{model}} + 2BC^2) + 9BCd_{\text{model}} = B\left(16Cd_{\text{model}} + 2C^2\right). \tag{14}$$

- **Additional activations:**
  - Final RMSNorm: $B \times C \times d_{\text{model}}$,
  - Output embedding (logits): $B \times C \times V$,
  - Cross-entropy on logits: $B \times C \times V$.

Thus, the total activation memory is:

$$\begin{aligned} M_{\text{act}} &= L\left(16BCd_{\text{model}} + 2BC^2\right) + BCd_{\text{model}} + 2BCV \\ &= B\left(L\left(16Cd_{\text{model}} + 2C^2\right) + Cd_{\text{model}} + 2CV\right). \end{aligned} \tag{15}$$

4.2.3 Gradients

Each parameter has an associated gradient (of the same shape), so:

$$M_{\text{grad}} = M_{\text{params}}.$$

4.2.4 Optimizer State

AdamW requires two additional copies of each parameter (for the first and second moment estimates), so:

$$M_{\text{opt}} = 2\, M_{\text{params}}.$$

4.2.4.1 Total Peak Memory

The total peak memory is the sum of parameter storage, activations, gradients, and optimizer state:

$$M_{\text{total}} = M_{\text{params}} + M_{\text{act}} + M_{\text{grad}} + M_{\text{opt}} = M_{\text{act}} + 4\, M_{\text{params}}. \tag{16}$$

$$\begin{aligned} M_{\text{total}} = 4 \times (&d_{\text{model}} \times C + d_{\text{model}} \times V + (12d_{\text{model}} + 2d_{\text{model}}) \times L + d_{\text{model}} \times V) \\ &+ B(L(16Cd_{\text{model}} + 2C^2) + Cd_{\text{model}} + 2CV) \end{aligned} \tag{17}$$

4.3 Memory for GPT-2 XL and Maximum Batch Size

For GPT-2 XL with $L = 48$, $d_{\text{model}} = 1600$, $V = 50257$, $C = 1024$:

$$M_{\text{params}} = 6,548,704,000 \text{ bytes} \tag{18}$$

$$M_{\text{act}} = 5,854,076,928 \text{ bytes per batch} \tag{19}$$

Since the GPU has 80 GB memory:

$$80,000,000,000 = 4 \times 6,548,704,000 + B_{\text{max}} \times 5,854,076,928 \tag{20}$$

Solving for B_{max}:

$$B_{\text{max}} = 9.191062 \tag{21}$$

So we can have a maximum batch size of 9 with the GPT-2 XL on a GPU with 80GB memory.

4.4 FLOPs for AdamW Update

The AdamW optimizer performs parameter updates using both momentum and adaptive learning rates. For each parameter θ with gradient g and optimizer states m and v, the update step involves a series of operations. We aim to compute the number of floating-point operations (FLOPs) required to perform one step of AdamW.

The number of FLOPs per step is derived based on the operations performed by AdamW during the update. The key steps include:

- **m-update:**
$$m \leftarrow \beta_1 m + (1 - \beta_1) g$$
This requires 2 multiplications and 1 addition, for a total of 3 FLOPs.

- **v-update:**
$$v \leftarrow \beta_2 v + (1 - \beta_2) g^2$$
This requires 1 multiplication for squaring g, 2 multiplications for scaling, and 1 addition, totaling 4 FLOPs.

- **Compute update term:** First, we compute $\sqrt{v}$ (1 FLOP), add ϵ (1 FLOP), divide m by the result (1 FLOP), and multiply by the learning rate (1 FLOP). This yields a total of 4 FLOPs.

- **Weight decay and parameter update:** The weight decay and parameter update require roughly 1–2 multiplications/subtractions, totaling 3 FLOPs.

Summing these gives:

$$3 + 4 + 4 + 3 = 14 \text{ FLOPs per parameter.}$$

For each parameter θ, AdamW performs approximately 14 floating-point operations. Therefore, the total number of FLOPs per step for AdamW is:

$$\text{FLOPs}_{\text{AdamW}} \approx 14\,P = 14\left[d_{\text{model}}\,C + 2\,d_{\text{model}}\,V + \left(12\,d_{\text{model}}^2 + 2\,d_{\text{model}}\right)\,L\right]\,.$$

$$\text{FLOPs}_{\text{AdamW}} \approx 14\,P = 26194816000.$$

4.5 Training Time with an A100

Total FLOPs per training step (assuming backwards pass has twice as much FLOPs than a foward pass):

$$F_{\text{step}} = 3F_{\text{fwd}} + F_{\text{AdamW}} \tag{22}$$

For GPT-2 XL we can use the previously calculated numbers:

$$F_{\text{step}} \approx 3 \times 3.5 \text{ TFLOPs} + 0.026 \text{ TFLOPs} \approx 10.05 \text{ TFLOPs} \tag{23}$$

Total training FLOPs for 400K steps:

$$F_{\text{train}} = 10.5T \times 400K \times 1024 \tag{24}$$

Assuming a peak throughput of 19.5 TFLOP/s on an NVIDIA A100 with 50% MFU:

$$\text{Effective Throughput} = 0.5 \times 19.5T = 9.75T \text{ FLOP/s} \tag{25}$$

Training time:

$$T_{\text{train}} = \frac{F_{\text{train}}}{9.75T} \tag{26}$$

$$T_{\text{train}} = \frac{10.5T \times 400K \times 1024}{9.75\text{TFLOP/s} \times 86400 \text{ seconds}} \tag{27}$$

$$T_{\text{train}} \approx 5105 \text{ days} \tag{28}$$

With the configurations it would take almost 14 years to train GPT-2 XL on a single A100. Because of this, the large scale language models are typically trained on multiple GPUs at a time. I believe that since the transformer architecture consists of multiple layers and heads it is easily parallelizable so in reality with many GPUs the training should be feasible.

5 Training the Transformer Language Model

Experimental configurations are as follows. All experiments are logged in wandb (Appendix B). Full training loop is in run_model.py

```
python ece496b_basics/run_model.py --experiment_name transformer-language-model \
--train_file data/tiny/train.npy --valid_file data/tiny/valid.npy \
--vocab_size 10000 \
--d_model 512 --num_heads 16 --num_layers 4 --d_ff 2048 \
--context_length 256 --batch_size 96 --total_iters 2000 \
--lr_max 0.001 --weight_decay 1e-2 \
--checkpoint_path checkpoints/checkpoint_${exp}-rtx5000.pth --save_interval 100 \
--log_interval 10 --eval_interval 100 --eval_iters 100 --ablation none
```

The general parameters for the transformer model were:

- $d_{\text{model}} = 512$: The dimensionality of the model's hidden representations.
- **num_heads** = 16: The number of attention heads in the multi-head self-attention mechanism.
- **num_layers** = 4: The number of transformer layers in the model.
- **d_ff** = 2048: The dimensionality of the feedforward network within each transformer layer.
- **context_length** = 256: The number of tokens in the input the model processes at a time.
- **lr_max** = 0.001: The maximum learning rate during training for cosine scheduling.

The hyperparameters chosen for the AdamW optimizer follow the standard from Kingma and Ba:

- **betas** = (0.9, 0.999): Exponential decay rates for the first and second moment estimates.
- **eps** = 1×10^{-8}: A small constant added for numerical stability.
- **weight_decay** = 1×10^{-2}: L2 regularization to prevent overfitting.

Batch size and **iterations** were chosen such that the model processes roughly 33 million tokens in training.

6 Experiments

6.1 Learning rate experiments

Figure 3 displays the logged results of experiments conducted with different learning rates. I performed a learning rate sweep across the range [1e-5, 1e-4, 2e-4, 5e-4, 1e-3, 5e-3] to capture both high and low values.

The optimal learning rate from these experiments is 1e-3, which yielded a validation loss of approximately 1.9. Lower learning rates, such as 1e-5, resulted in much slower convergence and a higher final loss. Conversely, the highest learning rate of 5×10^{-3} led to increased instability, particularly visible in the training loss curve (Figure 2).

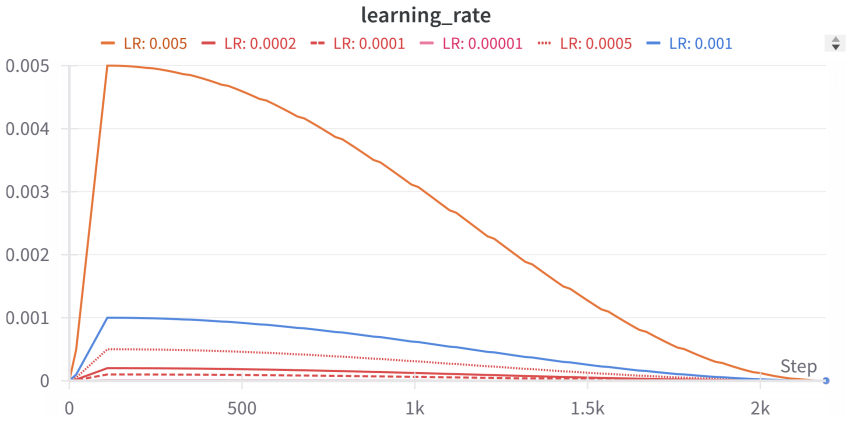

Figure 1: Different max learning rates with cosine scheduling.

Figure 2: Effects of learning rate on training loss.

Figure 3: Effects of learning rate on validation loss

6.2 Batch Size Modifications

I performed tests on batch sizes of $1, 2, 8, 32, 64, 96$. A batch size of 128 resulted in an out-of-memory error on the NVIDIA RTX 5000.

As seen in Figures 4 and 5, higher batch sizes resulted in lower loss values and increased stability. The training loss plot indicates that a batch size of 1 produced an unstable loss curve, whereas a batch size of 96 was significantly smoother. Higher batch sizes could result in more stability because they provide a better approximation of the true gradient during optimization.

However, this stability comes at a cost. Figure 6 shows the time per iteration (with a running average to smooth out evaluation iterations). As expected, increasing the batch size leads to longer iteration times.

Figure 4: Validation loss for different batch sizes.

Figure 5: Training loss for different batch sizes.

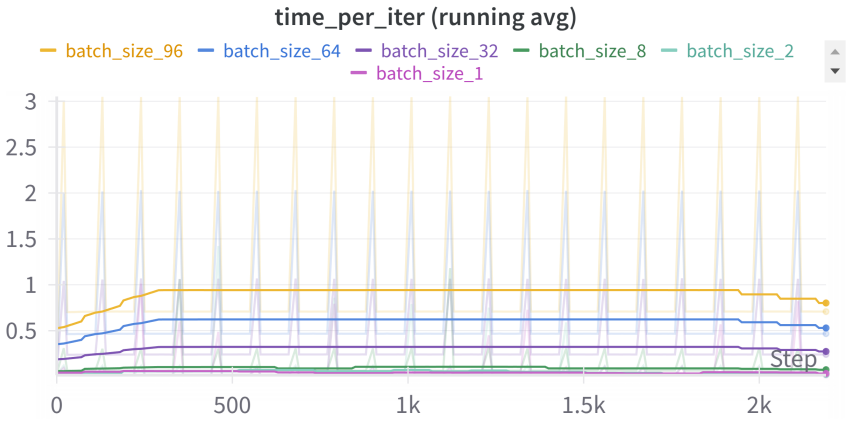

Figure 6: Time per iteration for different batch sizes.

6.3 Generation on Tiny Stories

I trained two models for generation on the TinyStories dataset. One of them was run on an NVIDIA RTX 5000, with a batch size of 96 training for 2000 iterations resulting in a total of 33 million tokens processed. The other was run on an NVIDIA H100 with a batch size of 256 training for 5000 iterations, resulting in a total of 330 million tokens processed.

After training the transformer models on the TinyStories dataset, I generated text using the trained checkpoint with the following parameters: temperature = 0.8 and top-p = 0.9. The results of the generated text were quite fluent, with the output maintaining some semblance of a coherent narrative structure and some degree of contextual relevance.

The output of the 330M token processed model shown in figure 8 has even more coherency and structure.

One key factor that might have contributed to the fluency is the structure of the TinyStories dataset itself. Since it consists of simple, short stories with a regular structure, the model was able to learn common patterns of storytelling, such as character introduction, action, conflict, and resolution. This consistency likely helped the model generate fluent and logical sequences of events, contributing to the high quality of the output.

However, since dataset is simple and consists of stories, the model output begins to deteriorate when trying to generalize to unseen sequences. We can see this in figure 9 and figure 10 when asked "What is the square root of pi?" The output loses it's coherence seen in the earlier prompts. Interestingly, the output of the 330 million token model seems to loosely output text related to geometry which is slightly relevant to the prompt.

```
Once upon a time, there was a big, heavy rock. It was very old and very small. It had a
    secret. The rock loved to play with the rock. It would rock, jump, and roll it all
    around.
One day, the rock met a little bug. The bug said, "Hi, I am Sam. What are you doing?" The
    rock said, "I am playing with you, little bug. I do not know how to jump too!"
Sam and the bug played together all day. They had so much fun. When it was time to go home,
    they said goodbye and went to their homes. They were happy they had a great day at the
    park.
<|endoftext|>
```

Figure 7: Output of model trained on TinyStories
(RTX5000, 33M tokens processed)
Prompt: "Once upon a time"

```
Once upon a time, there was a little girl named Lily. She loved to play outside in the sun.
    One day, she saw a big, pretty tree. The tree had many colors like red, orange, and
    yellow. Lily wanted to play near the tree, but she knew it was not safe.
Lily had an idea. She would pick some of her favorite colors to make the tree look pretty.
    She put the colors on the tree and started to pick the colors. As she picked the colors,
    she saw a little boy named Tim. Tim was sad because he had no color.
Lily thought about what to do. She remembered the tree with the colorful leaves. She went to
    Tim and said, "I found this tree. It is not too bad." Tim looked at the tree and saw
    that it was not big, but it was still beautiful. Lily and Tim became friends and played
    together every day.
<|endoftext|>
```

Figure 8: Output of model trained on TinyStories
(H100, 330M tokens processed).
Prompt: "Once upon a time"

```
What is the square root of pi? It's not a toy, it's a long tail. It is a fake is for my mom.
    She loves her. She loves her mom and dad. She gives her a hug and a kiss. She says she
    is a good toy.
She puts the sticker on her face and hugs. She says she is a good dog. She says she is a dog
    who likes to play with her. She says she loves her mom and dad. She says she loves her.
    She says she loves her mom and dad. She says she loves her. She says she can do that she
    loves her mom and dad. She says she is very smart and kind. She says she loves her too.
    She says she loves her mom and dad.
She says she is a good dog. She says she is very kind and brave. She says she loves her mom
    and dad. She says she likes her too. She says she will love her mom and dad. She says
    she loves her too. She says she loves her mom and dad and her love. She says she loves
    her mom and dad. She says she loves her too.
<|endoftext|>
```

Figure 9: Output of model trained on TinyStories
(RTX5000, 33M tokens processed).
Prompt: "What is the square root of pi?"

```
 What is the square root of pi? Why is it so big?"
Mom smiled and said, "The square is not a normal shape, but it is not really a round thing.
    It is a big square. It is a very big square that can fit in the square. Do you want to
    see?"
Sara nodded and took a step closer. She saw that the square was next to a big square. She
    wondered what it was. She wondered what it could be. She asked her mom, "What is it?"
Mom said, "It is a square, Sara. It is a square that is very small and very long. It has
    many sides and lines and sounds. It is a square for the square."
Sara thought for a moment and said, "Can I see it, mom?"
Mom said, "Yes, you can. But be careful, it is a triangle. It is very big and heavy. Don't
    touch it."
Sara took the triangle and looked at it closely. She saw that it was not a triangle, but a
    triangle. She saw that it was not a square, but a triangle. It was a globe that could
    talk and walk and talk. Sara smiled and said, "Hello, triangle.
```

Figure 10: Output of model trained on TinyStories
(H100, 330M tokens processed).
Prompt: "What is the square root of pi?"

7 Ablations

7.1 Parallel Layers

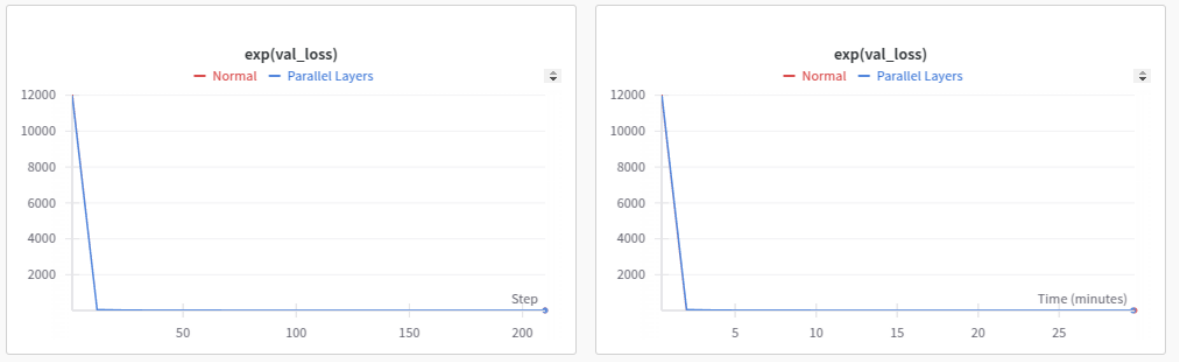

Figure 11: Parallel Layers in comparison to normal

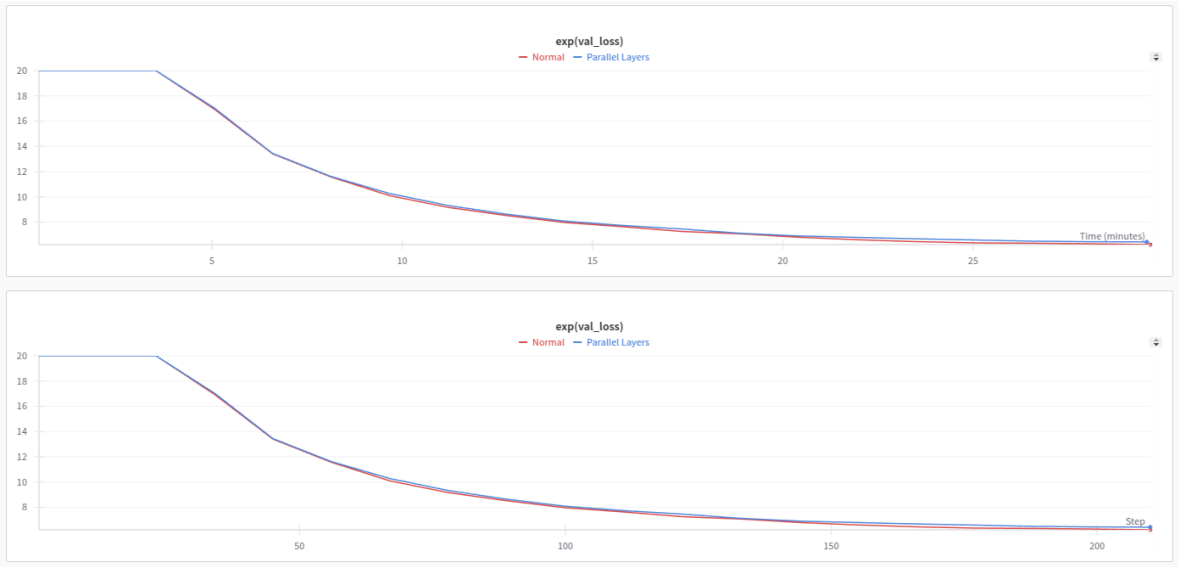

Figure 12: Parallel Layers perplexity zoomed in

When comparing the normal architecture to the parallel layers (Figure 11), there is not a huge difference in the loss. Only upon zooming in can we see that the normal architecture has a slightly lower perplexity (Figure 12), indicating slightly better performance; however, the difference is very small. The final perplexity of the normal transformer is **6.23**, while the perplexity of the parallel layer is **6.44**. When comparing the wall clock time, the parallel layer only had a slight decrease in the amount of time taken: **29.56** minutes for parallel layers and **29.65** minutes for the normal architecture.

One reason for the lack of difference is the limited depth in this model. Since it only processed around 33 million tokens and trained for 2000 iterations, the difference may be smaller and less observable. Perhaps with more training iterations or a deeper architecture, the differences between the two methods would be more pronounced. Based on this experiment, when choosing between parallel layers and normal layers, one must consider the specific model depth, the size of the dataset, and the desired training time. If computational resources are limited, parallel layers might not yield significant advantages unless applied in more complex, larger models. I would like to see if the results of applying parallel layers to a larger model with more resources.

7.2 Normalization Ablation

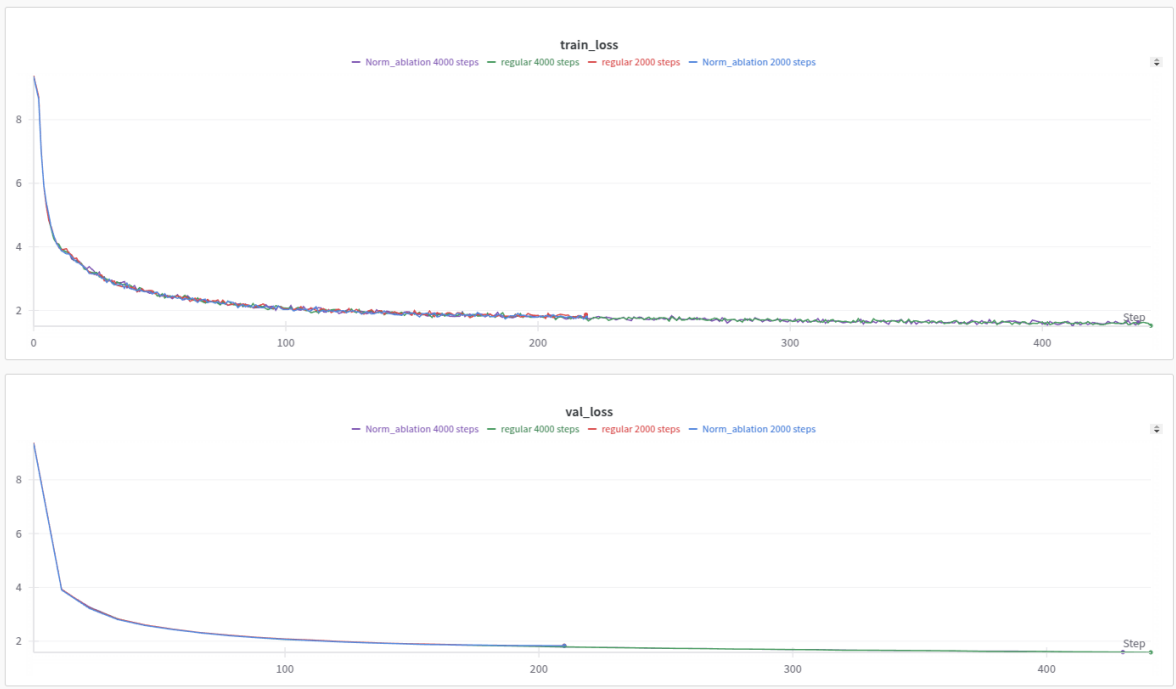

Figure 13: Normaliation ablation results

The results of removing all normalization did not affect the results as much as I expected. As shown in Figure 13, the two curves—one with normalization and one without—are very similar, indicating that the absence of normalization did not significantly change the training behavior or final performance. Due to this, I also attempted to train the model for more iterations to see if a divergence would occur, but even after extending the training duration, the results remained relatively stable.

7.3 Post-Norm Transformer

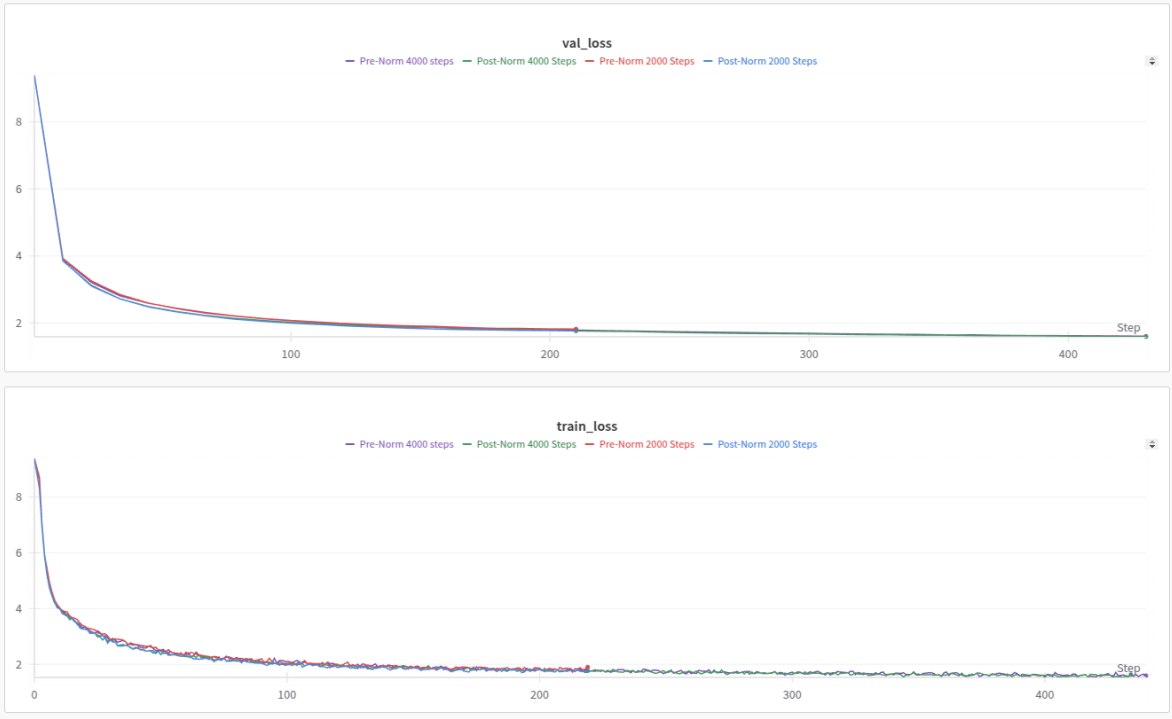

Figure 14: Post-Norm vs Pre-Norm

Similarly, there is not a substantial difference between the post-norm transformer and the pre-norm transformer in terms of performance. I also tried to run this experiment for an extended number of iterations. The post-norm model does not show any clear advantage over the pre-norm version in this experiment. However, I suspect that this result is due to the relatively shallow depth of the model used in this experiment. In deeper models, where the network relies more heavily on residual connections, the post-norm architecture may struggle to retain clean residuals, as it does not normalize before adding the residuals back into the output. It is possible that these runs would have diverged if we continued to train for several hours.

7.4 OWT Training

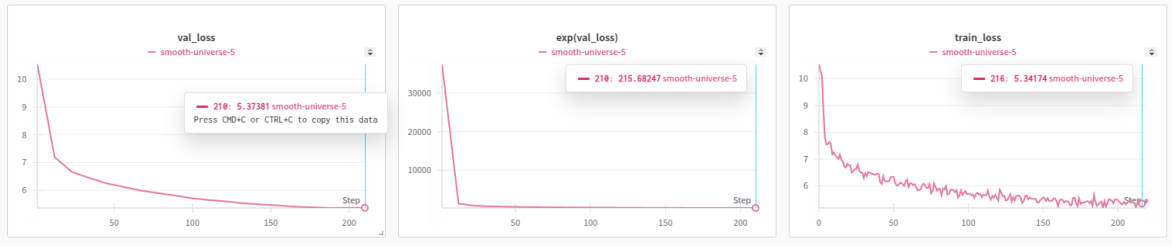

Figure 15: Results of OWT training

There were a few challenges when training the language model on the OpenWebText (OWT) dataset. Due to memory constraints on the RTX 5000 GPU, I was forced to lower the batch size from 96 (as used in the TinyStories experiment) to 32, in order to complete training. Unfortunately, I was also unable to run this experiment with the H100 due to resource unavailability at the time of testing.

The generated text from the OWT-trained model is noticeably less fluent and coherent compared to the output produced using TinyStories. This is also conflated with the higher end loss and perplexity

values when compared to the TinyStories dataset.

When prompted with the same phrases like "Once upon a time" or "What is the square root of pi?", the resulting outputs are fragmented and less consistent in narrative structure. Several factors contribute to this difference in quality:

- **Dataset Characteristics:** OpenWebText is a more varied and less structured dataset compared to TinyStories, which consists of short stories with similar narrative patterns. The diversity and noise present in OWT make it harder for the model to learn coherent language patterns.
- **Reduced Batch Size and Training Iterations:** The necessity to lower the batch size to 32, combined with maintaining the same number of training iterations as in the TinyStories experiment, effectively reduced the compute budget. This likely contributed to suboptimal convergence and poorer output quality.

```
 Once upon a time of the attacks.

The vast majority of the following day, after it was perhaps responsible for the time of the
    urban world.

The Native Democrat, also claims that the outcome of the government's ruling, not a
    second-cost strategic currency, the government's government, so it is not the head of
    the government's policies of a platform.

"The military is a very good reason for all American people are talking to the state. These
    are not going to happen."

But if someone else and Mr. his case he never made him because he was a particularly
    difficult for any particular issues.

"I think he'd actually been in this context," he said. "I believe that he is a foreign law
    of what it is."

He said: "They should have been a bigger about. I would never say, as I say, "I'm not to say
    something's the people who is, but it is not a moment of," he said.

The bill of the police have been also known as many of the idea, but the reason people's
    support for this, since the country's constitutional policy.

"It's very difficult to
```

Figure 16: Output of model trained on OpenWebText
(RTX5000, 33M tokens processed).
Prompt: "Once upon a time"

```
What is the square root of pi?

The three athletes who hold in the country's legacy of state school district, as well as the
    "silt" of the legalization of the country's largest power is not to be a larger business
    or nothing.

The University of the American Revolution of State University is a parallel facility in the
    Netherlands with the Great Depression of the U.S. Senate, and one of its members.

A mission of the possibility of the victims of the Iraqi government, which the government is
    not simply a threat of the United States, the U.S. Department of Justice, said: "If the
    law they will be in an email to the money that we have, they are like a sense of how
    government cannot become a government to be in a difficult way, it's clear that it is
    probably not to be willing to leave the project," he said.

The US government also urged the first-largest system will be paid to vote for the US, but
    it is the first is important to be able to do with the collapse of the US.

"The government of a cultural structure is a case to vote in the government's environment, a
    growing capacity and the Iraq, because there is
```

Figure 17: Output of model trained on OpenWebText
(RTX5000, 33M tokens processed).
Prompt: "What is the square root of pi?"

Appendix

A Code Availability

All code mentioned in this report is available at https://github.com/eve-liya/assignment1-basics.

B Wandb Logging

All logs of the transformer model experiments can be found on my Wandb projects:

- **Learning rate experiment logs:** learning_rate_sweep-rtx5000
- **Batch size experiment logs:** batch_size-exp-rtx5000
- **Parallel layer experiment logs:** parallel-rtx5000-1
- **Norm ablation experiment logs:** no_norm-rtx5000-1
- **Post-Norm ablation experiment logs:** post_norm-rtx5000-1
- **OWT training logs:** OWT
- **TinyStories H100 experiment logs:** H100-exp