CS 336: Assignment 1

Brandon Snider

April 15, 2025

Contents

2. BPE Tokenizer

Problem (`unicode1`): Understanding Unicode (1 point)

a) `chr(0)` returns `'\x00'`

b) The string representation (`__repr__()`) shows the escape sequence (`'\x00'`), while the printed representation shows no visible output.

c) In the printed representation (within a call to `print`), this character produces no visible output; in a string representation (the first and third examples), the escape sequence `\x00` appears in the ouptut.

Problem (`unicode2`): Unicode Encodings (3 points)

a) By starting from a tiny initial vocabulary (256 byte values) and learning efficient merges based on data, we produce a compact, efficient vocabulary. We can then fit more semantic content into a fixed-size context window.

b) The function treats each byte as a Unicode character, but UTF-8 represents characters as sequences of 1-4 bytes. The string `"¢"` would be decoded incorrectly, because the single character's UTF-8 representation is 2 bytes.

c) `0x80 0x80`, because `0x80` (equivalently `10000000`) is a continuation byte, and cannot be used as the first byte in a unicode sequence.

Problem (`train_bpe`): BPE Tokenizer Training (15 points)

See `cs336_basics/train_bpe.py`

Problem (`train_bpe_tinystories`): BPE Training on TinyStories (2 points)

a) Time: 135.32s (0.038h)

   Memory: 4GB (per Scalene)

   Longest token: `' accomplishment'`. This makes sense. With a fairly large vocabulary and a dataset of clean English text, one would expect the longest tokens to be long strings of valid English that appear contiguously in the dataset.

b) Pre-tokenization took roughly half of the overall training time (102s). The specific bottleneck is creating a bytes object for each individual character in each regex match to construct the keys in the table of coarse-grained tokens.

Problem (`train_bpe_expts_owt`): BPE Training on OpenWebText (2 points)

a) Longest token:

```
b'\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82
\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82
\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82'
```

   which decodes to `ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ`

This makes sense. This repeated pattern is common when documents are double-encoded or improperly decoded, which is common in scraped web content. In fact, this exact byte sequence appear over 4,500 times in the OWT training set.

b) The OpenWebText tokenizer achieves a greater compression ratio, but with the tradeoff of having a much larger vocabulary size that enables it to capture domain-specific patterns and web content artifacts. The TinyStories tokenizer specializes in clean, simple English, reflecting the characteristics of its clean, narrow training set in contrast to the diverse, noisy content from the broader internet.

Problem (`tokenizer`): Implementing the tokenizer (15 points)

See `cs336_basics/tokenizer.py`

Problem (`tokenizer_experiments`): Experiments with Tokenizers (4 points)

a) TinyStories tokenizer compression ratio (bytes/token): 4.01

OpenWebText tokenizer compression ratio (bytes/token): 4.50

b) OpenWebText sample, tokenized with TinyStories tokenizer: 3.40

The compression ratio is significantly worse than the compression ratio that the same tokenizer achieves on a sample of data from the same distribution on which the tokenizer was trained. Specifically, the OpenWebText/TinyStories compression ratio is $\sim 85\%$ of the TinyStories/TinyStories compression ratio.

c) Throughput $\approx 6.8 \times 10^6$ bytes/second $= 6.8$ MB/second

$T_{\text{Pile}} \approx (825 \times 10^9)/(6.8 \times 10^6) = 121,324$ seconds ≈ 33.7 hours

d) `uint16` is appropriate because of our vocabulary sizes. Both vocabulary sizes are $> 2^8$ and $< 2^{16}$. This means we can't use an 8-bit representation (we'd have token IDs greater than the representation can store) and we don't need more than a 16-bit representation (all token IDs can be expressed in a 16-bit representation). `uint16` is therefore the most memory-efficient choice.

3. Transformer Language Model Architecture

Problem (`linear`): Implementing the linear module (1 point)

See `Linear` class in `cs336_basics/model.py`

Problem (`embedding`): Implementing the embedding module (1 point)

See `Embedding` class in `cs336_basics/model.py`

Problem (`rms_norm`): Root Mean Square Layer Normalization (1 point)

See `RMSNorm` class in `cs336_basics/model.py`

Problem (`positionwise_feedforward`): Position-wise FFN (2 points)

See `SwiGLU` class in `cs336_basics/model.py`

Problem (`rope`): Implement RoPE (2 point)

See `RotaryPositionalEmbedding` class in `cs336_basics/model.py`

Problem (`softmax`): Implement softmax (1 point)

See `softmax` function in `cs336_basics/model.py`

Problem (`scaled_dot_product_attention`): Implement scaled dot-product attention (5 points)

See `scaled_dot_product_attention` function in `cs336_basics/model.py`

Problem (`multihead_self_attention`): Implement causal multi-head self-attention (5 points)

See `CausalMultiHeadSelfAttention` class in `cs336_basics/model.py`

Problem (`transformer_block`): Implement the transformer block (3 points)

See `Block` class in `cs336_basics/model.py`

Problem (`transformer_lm`): Implement the Transformer LM (3 points)

See `Transformer` class in `cs336_basics/model.py`

Problem (`transformer_accounting`): LM resource accounting (5 points)

a) Expression for the total parameter count:

$$p_{\text{total}} = p_{\text{embedding}} + p_{\text{layers}} + p_{\text{ln\_final}} + p_{\text{lm\_head}} \tag{1}$$

Parameter count of the embedding matrix:

$$p_{\text{embedding}} = \text{vocab\_size} \times d_{\text{model}} = 50,257 \times 1,600 = 80,411,200 \tag{2}$$

Parameter count of all transformer blocks (un-gated MLP):

$$
\begin{aligned}
p_{\text{layers}} &= \text{num\_layers} \times p_{\text{layer}} \\
p_{\text{layer}} &= (2 \times p_{\text{ln}}) + p_{\text{attn}} + p_{\text{ffn}} \\
&= (2 \times d_{\text{model}}) + \left(p_{\text{wqkv}} + p_{\text{out\_proj}}\right) + (p_{\text{w1}} + p_{\text{w2}} + p_{\text{w3}}) \\
&= (2 \times d_{\text{model}}) + (d_{\text{model}} \times 3 \times d_{\text{model}} + d_{\text{model}} \times d_{\text{model}}) + (2 \times d_{\text{model}} \times d_{\text{ff}}) \\
&= (2 \times 1,600) + (1,600 \times 3 \times 1,600 + 1,600 \times 1,600) + (2 \times 1,600 \times 6,400) \\
&= 30,723,200 \\
p_{\text{layers}} &= 48 \times 30,723,200 \\
&= 1,474,713,600
\end{aligned}
\tag{3}
$$

Parameter count of the final layer norm:

$$
p_{\text{ln}} = d_{\text{model}} = 1600 \tag{4}
$$

Parameter count of the LM head (assuming no weight tying):

$$
p_{\text{lm\_head}} = d_{\text{model}} \times \text{vocab\_size} = 1,600 \times 50,257 = 80,411,200 \tag{5}
$$

Final parameter count:

$$
\begin{aligned}
p_{\text{total}} &= p_{\text{embedding}} + p_{\text{layers}} + p_{\text{ln\_final}} + p_{\text{lm\_head}} \\
&= 80,411,200 + 1,474,713,600 + 1600 + 80,411,200 \\
&= 1,635,537,600
\end{aligned}
\tag{6}
$$

Assuming each parameter is represented using single-precision floating point (4 bytes), the memory required to load the model is:

$$
\begin{aligned}
\text{memory} &= p_{\text{total}} \times \text{memory\_per\_param} \\
&= 1,635,537,600 \times 4 \\
&= 6,542,150,400 \text{ bytes} \\
&\approx 6.54\text{GB}
\end{aligned}
\tag{7}
$$

b) **GPT-2 XL analysis:**

FLOPs to compute queries, keys, and values across all layers (using a W_{qkv} matrix):

$$
\begin{aligned}
F_{\text{attn\_qkv}} &= \text{num\_layers} \times [2 \times d_{\text{model}} \times (3 \times d_{\text{model}}) \times \text{context\_length}] \\
&= 48 \times [2 \times 1,600 \times (3 \times 1600) \times 1,024] \\
&= 754,974,720,000
\end{aligned}
\tag{8}
$$

Computing attention weights ($Q^T K$) across all layers:

$$
\begin{aligned}
F_{\text{attn\_weights}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times d_{\text{model}} \times \text{context\_length}) \\
&= 48 \times (2 \times 1,024 \times 1,600 \times 1,024) \\
&= 161,061,273,600
\end{aligned}
\tag{9}
$$

Computing attention values (WV, where W represents normalized attention weights):

$$
\begin{aligned}
F_{\text{attn\_values}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times \text{context\_length} \times d_{\text{model}}) \\
&= 48 \times (2 \times 1,024 \times 1,024 \times 1,600) \\
&= 161,061,273,600
\end{aligned} \tag{10}
$$

Output projection after the attention operation, across all layers:

$$
\begin{aligned}
F_{\text{attn\_out}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{model}} \times \text{context\_length}) \\
&= 48 \times (2 \times 1,600 \times 1,600 \times 1,024) \\
&= 251,658,240,000
\end{aligned} \tag{11}
$$

Up projection in the FFN, across all layers:

$$
\begin{aligned}
F_{\text{ffn\_up}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{ff}} \times \text{context\_length}) \\
&= 48 \times (2 \times 1,600 \times 6,400 \times 1,024) \\
&= 1,006,632,960,000
\end{aligned} \tag{12}
$$

Down projection in the FFN, across all layers (same m, n, p; different order):

$$
F_{\text{ffn\_down}} = F_{\text{ffn\_up}} = 1,006,632,960,000 \tag{13}
$$

Final output projection (LM head):

$$
\begin{aligned}
F_{\text{lm\_head}} &= 2 \times \text{vocab\_size} \times d_{\text{model}} \times \text{context\_length} \\
&= 2 \times 50,257 \times 1,600 \times 1,024 \\
&= 164,682,137,600
\end{aligned} \tag{14}
$$

Total FLOPs:

$$
\begin{aligned}
F_{\text{total}} &= F_{\text{attn}} + F_{\text{ffn}} + F_{\text{lm\_head}} \\
F_{\text{attn}} &= F_{\text{attn\_qkv}} + F_{\text{attn\_weights}} + F_{\text{attn\_values}} + F_{\text{attn\_out}} \\
&= 754,974,720,000 + (2 \times 161,061,273,600) + 251,658,240,000 \\
&= 1,328,755,507,200 \\
&\approx 1.33 \times 10^{12} \text{ FLOPs} \\
F_{\text{ffn}} &= F_{\text{ffn\_up}} + F_{\text{ffn\_down}} \\
&= 2 \times 1,006,632,960,000 \\
&= 2,013,265,920,000 \\
&\approx 2.01 \times 10^{12} \text{ FLOPs} \\
F_{\text{total}} &= 1,328,755,507,200 + 2,013,265,920,000 + 164,682,137,600 \\
&= 3,506,703,564,800 \\
&\approx 3.51 \times 10^{12} \text{ FLOPs}
\end{aligned} \tag{15}
$$

Proportions:

$$
\begin{aligned}
P_{\text{attn\_qkv}} &\approx 21.53\% \\
P_{\text{attn\_weights}} &\approx 4.59\% \\
P_{\text{attn\_values}} &\approx 4.59\% \\
P_{\text{attn\_out}} &\approx 7.18\% \\
P_{\text{attn}} &\approx 37.89\% \\
\\
P_{\text{ffn\_up}} &\approx 28.71\% \\
P_{\text{ffn\_down}} &\approx 28.71\% \\
P_{\text{ffn}} &\approx 57.41\% \\
\\
P_{\text{lm\_head}} &\approx 4.70\%
\end{aligned}
\tag{16}
$$

c) The FFNs require the most FLOPs by far, accounting for roughly 57% of the total (with each of the three matrix multiplications in the FFNs contributing equally). The attention blocks are the next most significant, accounting for roughly 38% of the total.

d) **GPT-2 small analysis:**

FLOPs to compute queries, keys, and values across all layers (using a W_{qkv} matrix):

$$
\begin{aligned}
F_{\text{attn\_qkv}} &= \text{num\_layers} \times [2 \times d_{\text{model}} \times (3 \times d_{\text{model}}) \times \text{context\_length}] \\
&= 12 \times [2 \times 768 \times (3 \times 768) \times 1,024] \\
&= 43,486,543,872
\end{aligned}
\tag{17}
$$

Computing attention weights ($Q^T K$) across all layers:

$$
\begin{aligned}
F_{\text{attn\_weights}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times d_{\text{model}} \times \text{context\_length}) \\
&= 12 \times (2 \times 1,024 \times 768 \times 1,024) \\
&= 19,327,352,832
\end{aligned}
\tag{18}
$$

Computing attention values (WV, where W represents normalized attention weights):

$$
\begin{aligned}
F_{\text{attn\_values}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times \text{context\_length} \times d_{\text{model}}) \\
&= 12 \times (2 \times 1,024 \times 1,024 \times 768) \\
&= 19,327,352,832
\end{aligned}
\tag{19}
$$

Output projection after the attention operation, across all layers:

$$
\begin{aligned}
F_{\text{attn\_out}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{model}} \times \text{context\_length}) \\
&= 12 \times (2 \times 768 \times 768 \times 1,024) \\
&= 14,495,514,624
\end{aligned}
\tag{20}
$$

Up projection in the FFN, across all layers:

$$
\begin{aligned}
F_{\text{ffn\_up}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{ff}} \times \text{context\_length}) \\
&= 12 \times (2 \times 768 \times 3,072 \times 1,024) \\
&= 57,982,058,496
\end{aligned} \tag{21}
$$

Down projection in the FFN, across all layers (same m, n, p; different order):

$$
F_{\text{ffn\_down}} = F_{\text{ffn\_up}} = 57,982,058,496 \tag{22}
$$

Final output projection (LM head):

$$
\begin{aligned}
F_{\text{lm\_head}} &= 2 \times \text{vocab\_size} \times d_{\text{model}} \times \text{context\_length} \\
&= 2 \times 50,257 \times 768 \times 1,024 \\
&= 79,047,426,048
\end{aligned} \tag{23}
$$

Total FLOPs:

$$
\begin{aligned}
F_{\text{attn}} &= F_{\text{attn\_qkv}} + F_{\text{attn\_weights}} + F_{\text{attn\_values}} + F_{\text{attn\_out}} \\
&= 43,486,543,872 + 19,327,352,832 + 19,327,352,832 + 14,495,514,624 \\
&= 96,636,764,160 \\
F_{\text{ffn}} &= F_{\text{ffn\_up}} + F_{\text{ffn\_down}} \\
&= 57,982,058,496 + 57,982,058,496 \\
&= 115,964,116,992 \\
F_{\text{total}} &= F_{\text{attn}} + F_{\text{ffn}} + F_{\text{lm\_head}} \\
&= 96,636,764,160 + 115,964,116,992 + 79,047,426,048 \\
&= 291,648,307,200 \\
&\approx 2.92 \times 10^{11} \text{ FLOPs}
\end{aligned} \tag{24}
$$

Proportions:

$$
\begin{aligned}
P_{\text{attn\_qkv}} &\approx 14.91\% \\
P_{\text{attn\_weights}} &\approx 6.63\% \\
P_{\text{attn\_values}} &\approx 6.63\% \\
P_{\text{attn\_out}} &\approx 4.97\% \\
P_{\text{attn}} &\approx 33.13\% \\
\\
P_{\text{ffn\_up}} &\approx 19.88\% \\
P_{\text{ffn\_down}} &\approx 19.88\% \\
P_{\text{ffn}} &\approx 39.76\% \\
\\
P_{\text{lm\_head}} &\approx 27.1\%
\end{aligned} \tag{25}
$$

GPT-2 medium analysis:

FLOPs to compute queries, keys, and values across all layers (using a W_{qkv} matrix):

$$
\begin{aligned}
F_{\text{attn\_qkv}} &= \text{num\_layers} \times [2 \times d_{\text{model}} \times (3 \times d_{\text{model}}) \times \text{context\_length}] \\
&= 24 \times [2 \times 1,024 \times (3 \times 1,024) \times 1,024] \\
&= 154,618,822,656
\end{aligned} \tag{26}
$$

Computing attention weights ($Q^T K$) across all layers:

$$
\begin{aligned}
F_{\text{attn\_weights}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times d_{\text{model}} \times \text{context\_length}) \\
&= 24 \times (2 \times 1,024 \times 1,024 \times 1,024) \\
&= 51,539,607,552
\end{aligned} \tag{27}
$$

Computing attention values (WV, where W represents normalized attention weights):

$$
\begin{aligned}
F_{\text{attn\_values}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times \text{context\_length} \times d_{\text{model}}) \\
&= 24 \times (2 \times 1,024 \times 1,024 \times 1,024) \\
&= 51,539,607,552
\end{aligned} \tag{28}
$$

Output projection after the attention operation, across all layers:

$$
\begin{aligned}
F_{\text{attn\_out}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{model}} \times \text{context\_length}) \\
&= 24 \times (2 \times 1,024 \times 1,024 \times 1,024) \\
&= 51,539,607,552
\end{aligned} \tag{29}
$$

Up projection in the FFN, across all layers:

$$
\begin{aligned}
F_{\text{ffn\_up}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{ff}} \times \text{context\_length}) \\
&= 24 \times (2 \times 1,024 \times 4,096 \times 1,024) \\
&= 206,158,430,208
\end{aligned} \tag{30}
$$

Down projection in the FFN, across all layers (same m, n, p; different order):

$$
F_{\text{ffn\_down}} = F_{\text{ffn\_up}} = 206,158,430,208 \tag{31}
$$

Final output projection (LM head):

$$
\begin{aligned}
F_{\text{lm\_head}} &= 2 \times \text{vocab\_size} \times d_{\text{model}} \times \text{context\_length} \\
&= 2 \times 50,257 \times 1,024 \times 1,024 \\
&= 105,396,568,064
\end{aligned} \tag{32}
$$

Total FLOPs:

$$
\begin{aligned}
F_{\text{attn}} &= F_{\text{attn\_qkv}} + F_{\text{attn\_weights}} + F_{\text{attn\_values}} + F_{\text{attn\_out}} \\
&= 154,618,822,656 + 51,539,607,552 + 51,539,607,552 + 51,539,607,552 \\
&= 309,237,645,312 \\
F_{\text{ffn}} &= F_{\text{ffn\_up}} + F_{\text{ffn\_down}} \\
&= 206,158,430,208 + 206,158,430,208 \\
&= 412,316,860,416 \\
F_{\text{total}} &= F_{\text{attn}} + F_{\text{ffn}} + F_{\text{lm\_head}} \\
&= 309,237,645,312 + 412,316,860,416 + 105,396,568,064 \\
&= 826,951,073,792 \\
&\approx 8.27 \times 10^{11} \text{ FLOPs}
\end{aligned}
\tag{33}
$$

Proportions:

$$
\begin{aligned}
P_{\text{attn\_qkv}} &\approx 18.70\% \\
P_{\text{attn\_weights}} &\approx 6.23\% \\
P_{\text{attn\_values}} &\approx 6.23\% \\
P_{\text{attn\_out}} &\approx 6.23\% \\
P_{\text{attn}} &\approx 37.39\% \\
\\
P_{\text{ffn\_up}} &\approx 24.93\% \\
P_{\text{ffn\_down}} &\approx 24.93\% \\
P_{\text{ffn}} &\approx 49.86\% \\
\\
P_{\text{lm\_head}} &\approx 12.75\%
\end{aligned}
\tag{34}
$$

GPT-2 large analysis:

FLOPs to compute queries, keys, and values across all layers (using a W_{qkv} matrix):

$$
\begin{aligned}
F_{\text{attn\_qkv}} &= \text{num\_layers} \times [2 \times d_{\text{model}} \times (3 \times d_{\text{model}}) \times \text{context\_length}] \\
&= 36 \times [2 \times 1,280 \times (3 \times 1,280) \times 1,024] \\
&= 362,387,865,600
\end{aligned}
\tag{35}
$$

Computing attention weights ($Q^T K$) across all layers:

$$
\begin{aligned}
F_{\text{attn\_weights}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times d_{\text{model}} \times \text{context\_length}) \\
&= 36 \times (2 \times 1,024 \times 1,280 \times 1,024) \\
&= 96,636,764,160
\end{aligned}
\tag{36}
$$

Computing attention values (WV, where W represents normalized attention weights):

$$
\begin{aligned}
F_{\text{attn\_values}} &= \text{num\_layers} \times (2 \times \text{context\_length} \times \text{context\_length} \times d_{\text{model}}) \\
&= 36 \times (2 \times 1,024 \times 1,024 \times 1,280) \\
&= 96,636,764,160
\end{aligned} \tag{37}
$$

Output projection after the attention operation, across all layers:

$$
\begin{aligned}
F_{\text{attn\_out}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{model}} \times \text{context\_length}) \\
&= 36 \times (2 \times 1,280 \times 1,280 \times 1,024) \\
&= 120,795,955,200
\end{aligned} \tag{38}
$$

Up projection in the FFN, across all layers:

$$
\begin{aligned}
F_{\text{ffn\_up}} &= \text{num\_layers} \times (2 \times d_{\text{model}} \times d_{\text{ff}} \times \text{context\_length}) \\
&= 36 \times (2 \times 1,280 \times 5,120 \times 1,024) \\
&= 483,183,820,800
\end{aligned} \tag{39}
$$

Down projection in the FFN, across all layers (same m, n, p; different order):

$$
F_{\text{ffn\_down}} = F_{\text{ffn\_up}} = 483,183,820,800 \tag{40}
$$

Final output projection (LM head):

$$
\begin{aligned}
F_{\text{lm\_head}} &= 2 \times \text{vocab\_size} \times d_{\text{model}} \times \text{context\_length} \\
&= 2 \times 50,257 \times 1,280 \times 1,024 \\
&= 131,745,710,080
\end{aligned} \tag{41}
$$

Total FLOPs:

$$
\begin{aligned}
F_{\text{attn}} &= F_{\text{attn\_qkv}} + F_{\text{attn\_weights}} + F_{\text{attn\_values}} + F_{\text{attn\_out}} \\
&= 362,387,865,600 + 96,636,764,160 + 96,636,764,160 + 120,795,955,200 \\
&= 676,457,349,120 \\
F_{\text{ffn}} &= F_{\text{ffn\_up}} + F_{\text{ffn\_down}} \\
&= 483,183,820,800 + 483,183,820,800 \\
&= 966,367,641,600 \\
F_{\text{total}} &= F_{\text{attn}} + F_{\text{ffn}} + F_{\text{lm\_head}} \\
&= 676,457,349,120 + 966,367,641,600 + 131,745,710,080 \\
&= 1,774,570,700,800 \\
&\approx 1.77 \times 10^{12} \text{ FLOPs}
\end{aligned} \tag{42}
$$

Proportions:

$$
\begin{aligned}
P_{\text{attn\_qkv}} &\approx 20.42\% \\
P_{\text{attn\_weights}} &\approx 5.45\% \\
P_{\text{attn\_values}} &\approx 5.45\% \\
P_{\text{attn\_out}} &\approx 6.81\% \\
P_{\text{attn}} &\approx 38.12\% \\
\\
P_{\text{ffn\_up}} &\approx 27.23\% \\
P_{\text{ffn\_down}} &\approx 27.23\% \\
P_{\text{ffn}} &\approx 54.46\% \\
\\
P_{\text{lm\_head}} &\approx 7.42\%
\end{aligned}
\tag{43}
$$

Analysis:

The FFN computations increasingly dominate as model size increases. The contribution from the LM head is significant (greater than the contribution from attention) at the smallest model size, and diminishes quickly as model size increases.

e) The total FLOPs required increases from 3.51×10^{12} to 1.33×10^{14}. The FLOPs for all operations except the attention operation increase linearly in the length of the context window (by a factor of 2^4, in this case). The FLOPs for the attention operation (both $Q^T K$ and WV, where W represents the normalized attention weights) increase quadratically in the length of the context window (by a factor of 2^8, in this case).

4. Training a Transformer LM

Problem (`cross_entropy`): Implement cross entropy (2 points)

See `cross_entropy_loss` function in `cs336_basics/loss.py`

Problem (`learning_rate_tuning`): Tuning the learning rate (1 point)

For learning rates of 1, 1e1, and 1e2, the loss decreases more quickly as the learning rate is increased (reaching 23.0 with lr=1, and 10^-23 with lr=100). With a learning rate of 1e3, the loss diverges, reaching 10^18 by iteration 10, indicating too large a learning rate.

Problem (`adamw`): Implement AdamW (2 points)

See `AdamW` class in `cs336_basics/adamw.py`

Problem (`adamwAccounting`): Resource accounting for AdamW (2 points)

a) We express the peak memory requirements in terms of:

$$\begin{aligned} V &= \text{vocab\_size} \\ N &= \text{num\_layers} \\ d &= d_{\text{model}} \\ d_{\text{ff}} &= 4d \\ h &= \text{num\_heads} \\ T &= \text{context\_length} \\ B &= \text{batch\_size} \end{aligned} \tag{44}$$

Parameters:

Embeddings: $V \times d$

Each of the N transformer blocks:

- 2x RMSNorm: $2d$
- MHA: $W_{\text{qkv}} + W_{\text{out}} = [d \times (3 \times d)] + d^2 = 4d^2$:
- FFN: $2 \times 4d^2 = 8d^2$
- Total: $12d^2 + 2d$

Final RMSNorm: d

LM head: $d \times V$

Total parameter count: $P = (2Vd) + N(12d^2 + 2d) + d$

Parameter memory:

$$\text{ParamMemory} = 4P \text{ bytes} \tag{45}$$

Optimizer State

Each parameter has a first moment and second moment, so:

$$\text{AdamMemory} = 2 \times (4P) = 8P \text{ bytes} \tag{46}$$

Gradient Memory

We hold one float per parameter, so:

$$\text{GradMemory} = 4P \text{ bytes} \tag{47}$$

Activation Memory

Each transformer block:

- RMSNorm results: $2 \times B \times T \times d$
- MHA:
  - QKV projections: $3 \times B \times T \times d$
  - Attention scores ($Q^T K$): $B \times h \times T \times T$
  - Softmax over attention scores: $B \times h \times T \times T$
  - Weighted sum of values: $B \times T \times d$
  - Output projection: $B \times T \times d$
- FFN:
  - W_1 output: $B \times T \times 4d = 4 \times B \times T \times d$
  - SiLU activation: $B \times T \times 4d = 4 \times B \times T \times d$
  - W_2 output: $B \times T \times d$
- Total: $16(BTd) + 2(BhT^2)$

Across all N blocks: $N(16BTd + 2BhT^2)$

Final RMSNorm: $B \times T \times d$

Output embedding (LM head): $B \times T \times V$

Cross-entropy on logits: $B \times T$

Total activation count: $A = N(16BTd + 2BhT^2) + (BTd) + (BTV) + (BT)$

$$\text{ActMemory} = 4A \text{ bytes} \tag{48}$$

Final Peak Memory Expression

$$\begin{aligned} \text{TotalMemory} &= \text{ParamMemory} + \text{AdamMemory} + \text{GradMemory} + \text{ActMemory} \\ &= 4P + 8P + 4P + 4A \\ &= 16P + 4A \text{ bytes} \\ &= 16\big[(2Vd) + N(12d^2 + 2d) + d\big] + \\ &\quad 4\big[N(16BTd + 2BhT^2) + (BTd) + (BTV) + (BT)\big] \end{aligned} \tag{49}$$

b) $\text{TotalMemory}(B) = (15,311,904,768 \times B) + 26,168,601,600 \text{ bytes} \approx 26 \text{ GB}$

We require $\text{TotalMemory}(B) \leq 80 \times 10^9$ bytes, $B \in \mathbb{Z}$, so:

$$\begin{aligned} (15,311,904,768 \times B) + 26,168,601,600 &\leq 80,000,000,000 \\ \Rightarrow B &\leq 3 \end{aligned} \tag{50}$$

With 80 GB available, and storing every intermediate value for every layer in float32, our maximum batch size is 3.

c) Per part a), the total parameter count is: $P = (2Vd) + N(12d^2 + 2d) + d$

To run a step of AdamW, we compute the matrix multiplications for both the forward and backward passes. With a context length of $T = 1024$ and a batch size of B, we then have $1024 \times B$ data points and P parameters.

Forward pass FLOPs: $2 \times (1024 \times B) \times P$

Backward pass FLOPs: $4 \times (1024 \times B) \times P$

Total FLOPs: $6 \times (1024 \times B) \times P = 6 \times (1024 \times B) \times \left[(2Vd) + N(12d^2 + 2d) + d\right]$

d) FLOPs/step:

$$\begin{aligned} F_{\text{step}} &= 6 \cdot (1024 \times 1024) \cdot \left[(2 \cdot 50257 \cdot 1600) + 48(12 \cdot 1600^2 + 2 \cdot 1600) + 1600\right] \\ &= 6 \cdot (1024 \times 1024) \cdot 1,635,537,600 \\ &= 10,289,912,846,745,600 \\ &= 10.29 \times 10^{15} \text{ FLOPs} \end{aligned} \tag{51}$$

Total FLOPs:

$$F_{\text{total}} = F_{\text{step}} \times 400,000 = 4,115,965,138,698,240,000,000 \tag{52}$$

Achieved FLOP/s:

$$F_{\text{achieved}} = 19.5 * 10^{12} * 50\% = 9.75 \times 10^{12} \text{ FLOPs/s} \tag{53}$$

Time to train:

$$\begin{aligned} t_{\text{train}} &= F_{\text{total}} / F_{\text{achieved}} \\ &= 4,115,965,138,698,240,000,000 / (9.75 \times 10^{12}) \\ &= 422,150,270.64\text{s} \\ &= 4,886 \text{ days} \end{aligned} \tag{54}$$

Problem (`learning_rate_schedule`): Implement cosine learning rate schedule with linear warmup (1 point)

See `lr_cosine_schedule` function in `cs336_basics/lr_schedule.py`

Problem (`gradient_clipping`): Implement gradient clipping (1 point)

See `gradient_clip` function in `cs336_basics/gradient_clip.py`

5. Training Loop

Problem (`data_loading`): Implement data loading (2 point)

See `get_batch` function in `cs336_basics/data_loader.py`

Problem (`checkpointing`): Implement model checkpointing (1 points)

See `save_checkpoint` and `load_checkpoint` functions in `cs336_basics/checkpointing.py`

Problem (`training_together`): Put it together (4 points)

See `cs336_basics/train.py`

The training script supports:

- Loading configurations from JSON or YAML files supplied as command-line arguments
- Arbitrary command-line overrides
- Logging to WandB (and to local files and the console)
- Checkpointing (and resumption)
- Loading data with `np.memmap`

6. Generating text

Problem (`decoding`): Decoding (3 points)

See `decode` function in `cs336_basics/decode.py`

7. Experiments

Problem (`experiment_log`): Experiment Logging (3 points)

I used WandB to log my experiments. For each run, I logged the entire run configuration (hyperparameters, model configuratin, dataset paths, etc.), as well as the losses, perplexity, learning rate, gradient norms, and throughput (tokens/second). I tagged some runs to be make them easy to find (e.g. “lr-sweep”, “pre-norm-ablation”, etc.).

I’ve linked WandB pages for each of the experiments below. Here is a consolidated list:

EXPERIMENT	WANDB LINK
Learning Rate Sweep	WandB Report°
Batch Size Experiments	WandB Report°
Layer Norm Ablation	WandB Report°
Pre-Norm Ablation	WandB Report°
RoPE Ablation	WandB Report°
SwiGLU Ablation	WandB Report°
OpenWebText Baseline	WandB Report°
Leaderboard Runs	WandB Report°

Problem (`learning_rate`): Tune the learning rate (3 points)

a)

Figure 1: Learning rate sweep on TinyStories

WandB Report°

Final losses:

LEARNING RATE	EVAL/LOSS
1e-5	8.188
1e-4	2.824
1e-3	1.459
3e-3	1.392
5e-3	1.379
6e-3	1.381
7e-3	Diverged
1e-2	Diverged

Search strategy:

I started with a log-spaced search of 1e-5 to 1e-2. 1e-5 and 1e-4 were much too low, and I saw divergence at 1e-2. Given that training was very stable at 1e-3 and my loss was still higher than the 1.45 reference, I guessed that the sweetspot would be slightly greater (rather than slightly smaller). I started searching up from 1e-3 to the point of divergence. I tried 3e-3, 5e-3, and 7e-3. I saw divergence at 7e-3, so tried 6e-3 just to narrow down the divergence point. I found that 6e-3 was stable, and therefore probably near the edge of stability.

b) The folk wisdom seems to be roughly accurate in my case, though I got negligibly better loss and more stable training at 5e-3 (1e-3 down from the highest stable learning rate that I tried). This suggests that being close to the edge is good, but perhaps it's not necessary to be right up against it, and stability is more easily achieved by pulling back fractionally.

Problem (`batch_size_experiment`): Batch size variations (1 point)

a)

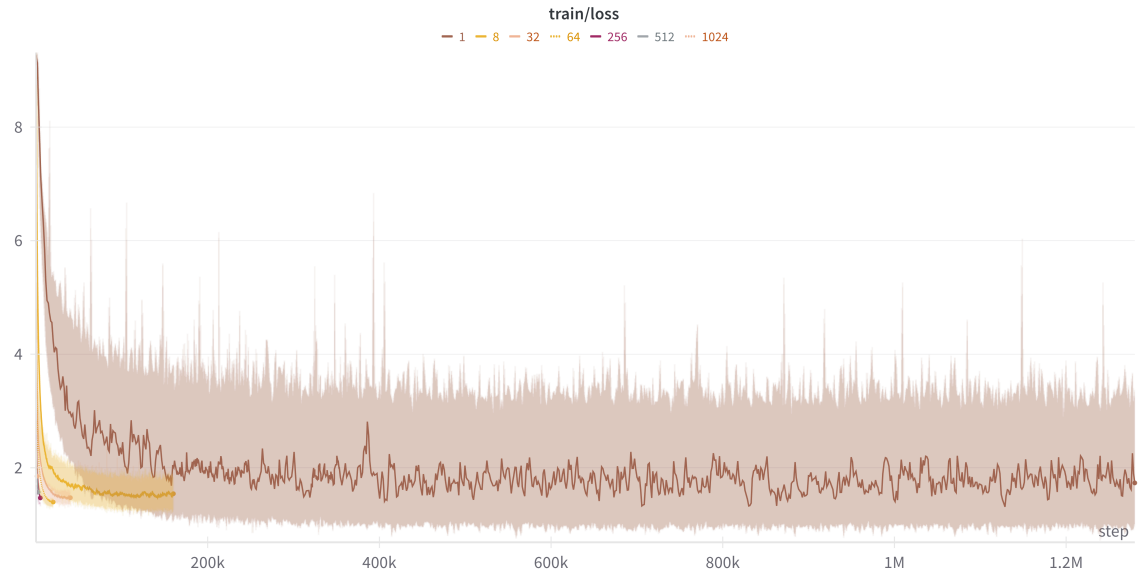

Figure 2: Batch size variations on TinyStories

WandB Report°

Findings:

- Throughput increases with batch size up to a point, after which it remains roughly flat despite increasing memory usage. At the smallest batch, training was extremely slow.
- Smaller batch sizes require lower learning rates to avoid divergence due to noisy gradients.
- There is a "sweet spot" range of batch sizes for a given token budget. Increasing batch size to the GPU limit (1024 in my case) degraded the quality of the final model, perhaps because there were too few gradient steps givne the fixed token budget.

Problem (`generate`): Generate text (1 point)

a) Decoding parameters:

```
print(decode(model, tokenizer, "The", max_new_tokens=512, temperature=0.7, top_p=0.9))
```

Generation:

> The squirrel said, "Hello, little bird! I have something for you." The little bird was very excited and said, "Thank you, Mr. Squirrel! I want to know what it is!" The squirrel took out a small piece of paper and gave it to the little bird.
>
> The little bird said, "This is a special paper. I will show you!" The squirrel took the paper and started to draw. The little bird was very happy to see the paper. The squirrel thanked the little bird and they became good friends. From that day on, they always played together in the forest, and the little bird always had a friend to help him when he needed it.

Comments:

The generation is fluent and coherent, and would fit well in the TinyStories dataset.

Temperature and top_p work together to control the diversity and determinism of the generations.

With a fixed temperature (e.g. 0.7), `top_p`, a smaller `top_p` (e.g. 0.1) narrows the pool of candidate tokens, and the generations come out very similar each time. A larger `top_p` (e.g. 0.99) allows for more diversity, but the generations can become low quality, losing coherence.

With a fixed `top_p`, increasing `temperature` flattens the probability distribution over the tokens in the candidate pool, and diversity increases. Decreasing `temperature` concentrates the probability distribution, and diversity decreases.

With a well-balanced temperature and top_p, the pool of candidate tokens is large enough to allow for diversity, but the probability distribution is concentrated enough to select unusual tokens with low probability, so generations can be both diverse and coherent.

Problem (`layer_norm_ablation`): Remove RMSNorm and train (1 point)

a)

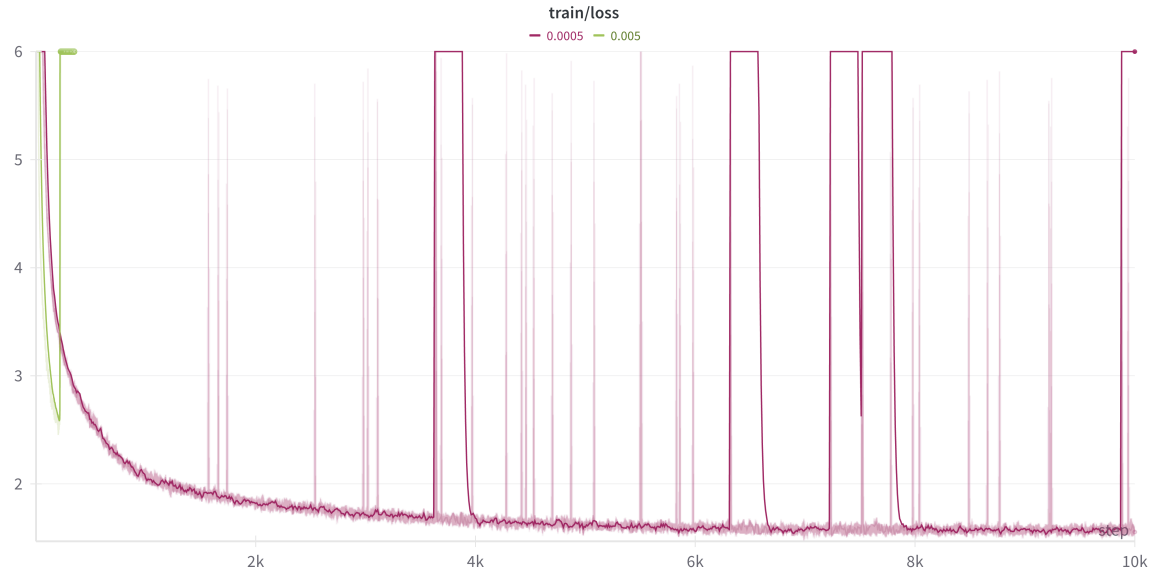

Figure 3: Layer norm ablation on TinyStories

WandB Report°

Comments:

Removing RMSNorms dramatically decreases stability. At the previous optimal learning rate, the optimizer diverges almost immediately. By decreasing the learning rate by 90%, I was able to train for full token budget, but still had significant spikes in loss, and a lower-quality final model (eval loss of 1.57 vs 1.38).

Problem (`pre_norm_ablation`): Implement post-norm and train (1 point)

a)

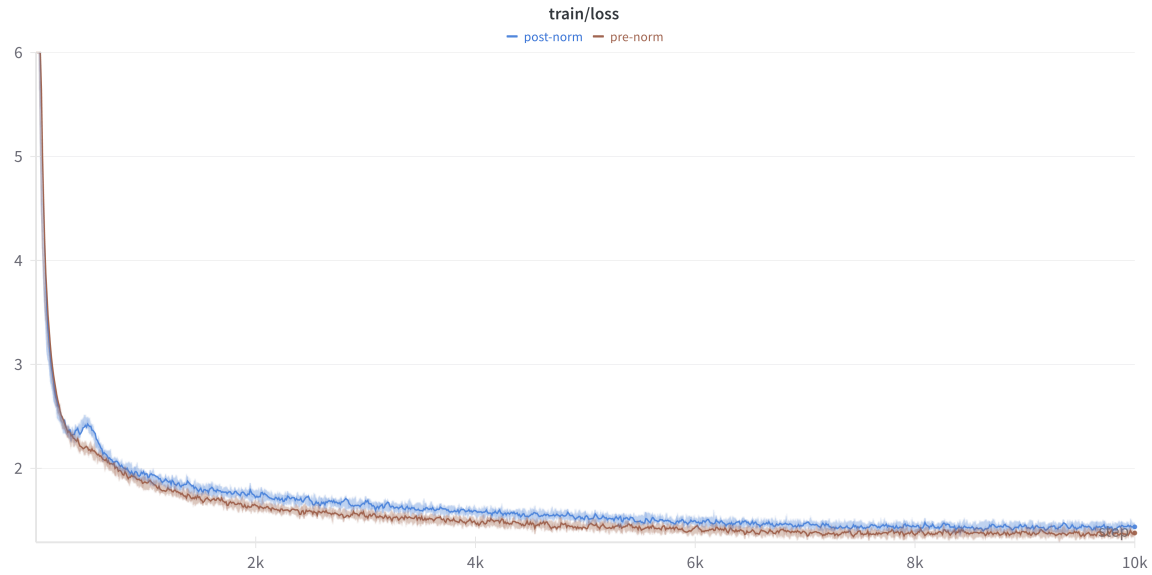

Figure 4: Pre-norm vs. post-norm on TinyStories

WandB Report°

Both training runs were quite stable. The pre-norm run produced a better final model (eval loss of 1.38 vs 1.44).

Problem (`no_pos_emb`): Implement NoPE (1 point)

a)

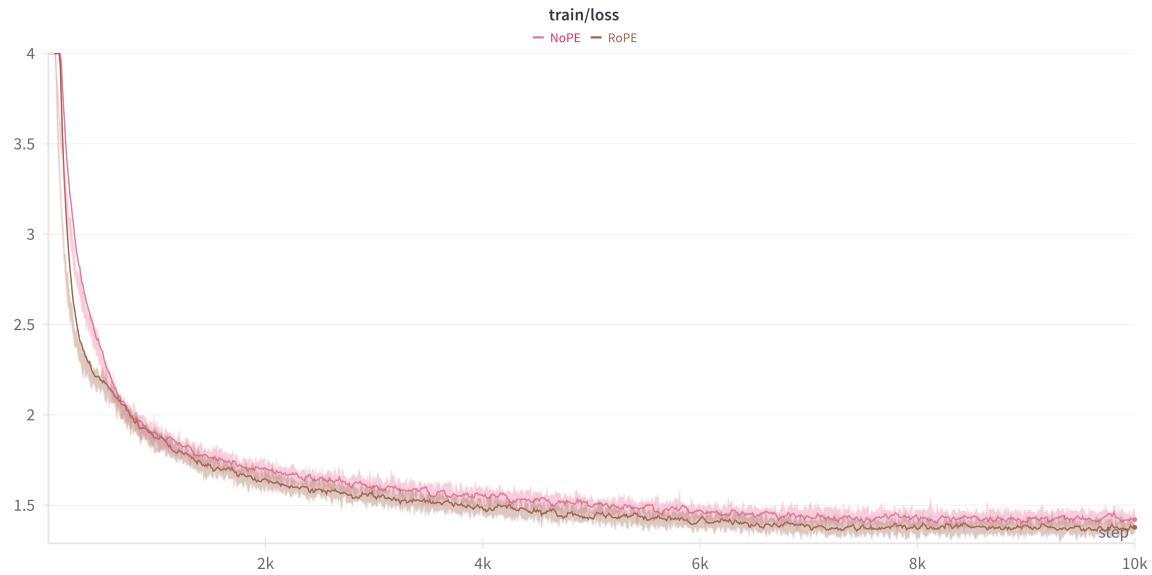

Figure 5: RoPE vs. NoPE on TinyStories

WandB Report°

RoPE helps, but the model is able to learn without it. The final loss is is not as good, perhaps because the ability to learn relationships that depend on knowledge of relative position is hampered. The final model with RoPE has eval loss of 1.38 vs 1.43 for the NoPE model.

Problem (`swiglu_ablation`): SwiGLU vs SiLU (1 point)

a)

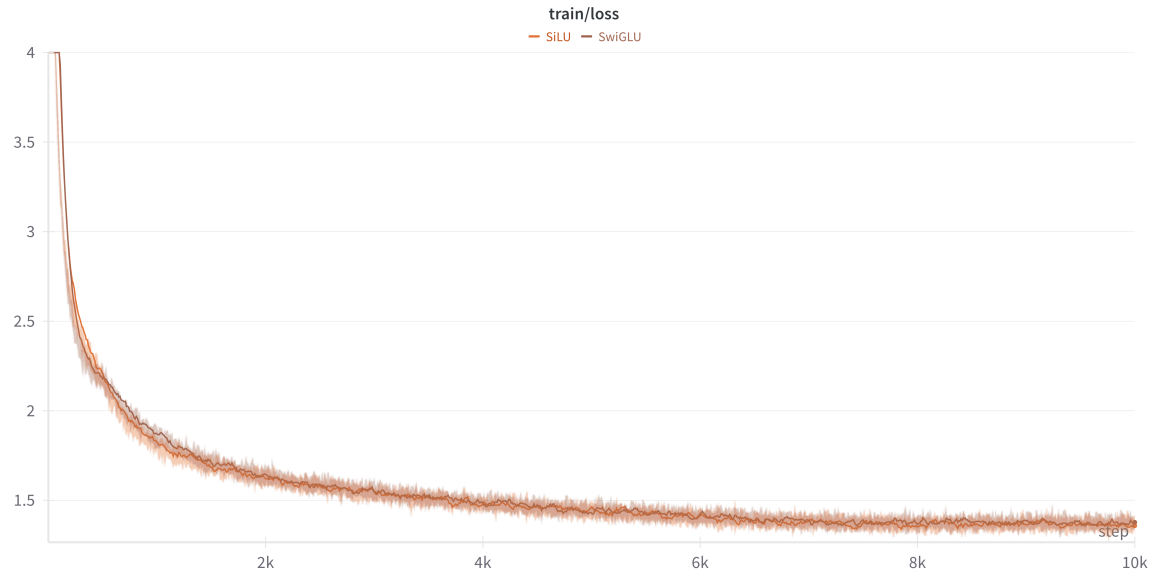

Figure 6: SwiGLU vs. SiLU on TinyStories

WandB Report°

Comments:

The two MLP variants perform almost identically, with SiLU actually producing every so slightly better eval loss (1.37 vs 1.38). My guess is that on a model and dataset of this size, this fairly small architectural difference is not particularly significant, and that observed differences at larger scaled just don't fully translate to these conditions.

Problem (`main_experiment`): Experiment on OWT (2 points)

a)

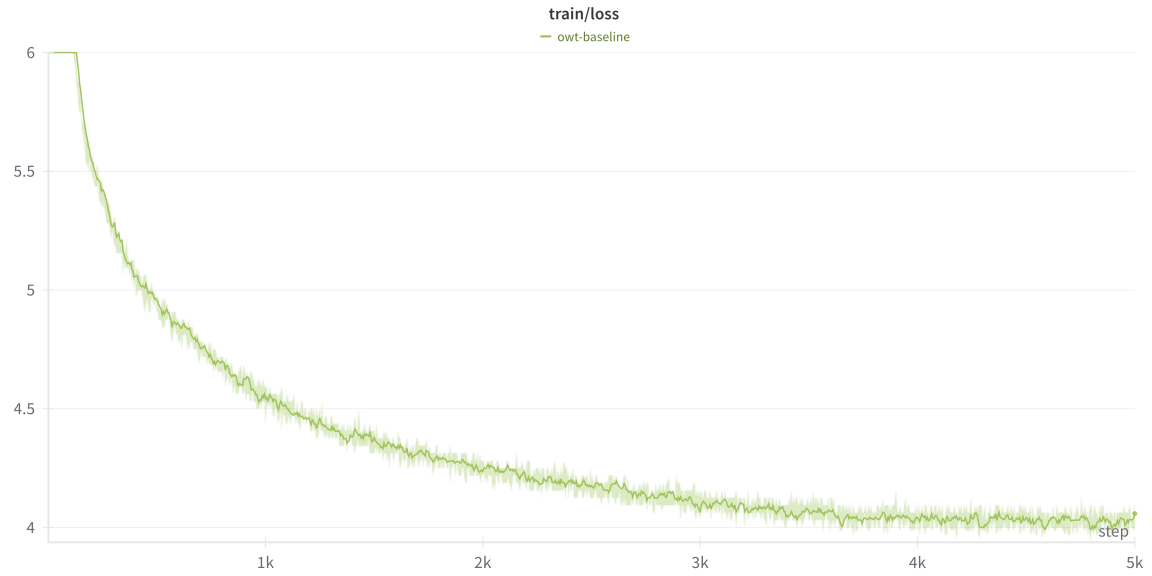

Figure 7: OpenWebText Learning Curve

WandB Report°

Comments:

Both training and validation losses are much higher than TinyStories (eval loss was 4.03 on OWT vs 1.38 on TinySTories). Concretely, that the probabilit distribution output by the model less closely matches the target distribution for the model trained on OWT compared to the model trained on TinyStories. More interpretively, the model trained on OWT has not learned the dataset as well as the model trained on TinyStories, so we should expect lower-quality generations.

Generation Prompt:

```
print(decode(model, tokenizer, "The", max_new_tokens=512, temperature=0.7, top_p=0.9))
```

Completion:

> The ‘King of the Mavs’ is a reference to the “King of the Mavs” of the Mavs, who were the two sons of the two brothers who are the ones who were a group of people, and the other the siblings, that they were part of the family. The two brothers are a man who was also the daughter of a man who was the first person who was a boy and was the first person to marry him.
>
> The son of the Mavs was born in the town of Mavs who was a man of honour. He was a woman who had been named the father of the Mavs who had been a boy in the Kiwi home in Sikuya, and the daughter of a grandfather who had been a child of a child.
>
> The son of a girl of the family is a woman of two. The family was also named as the daughter of the son of the daughter of the boy, who had been born in the same family.
>
> The family is named after the Sultan of Mavs’ son of Ravi, the son of a prince, who has been a household name for three years.
>
> The father of the boy, who had been a father of two sisters of his son, who lived in Sikuya, an uncle of his uncle and his mother, and two cousins of his father, Josephine, and daughter of Prince of Mavs.
>
> The son of Mavs’ son, the daughter of a child of a man who had been a father of two and a half years old, was a mother of three. He was a father of four.
>
> The son of the daughter of the prince, the daughter of the son of a child who was born in the mother of the father, was also a father of two siblings of the same family.
>
> “He was born in the father of the father of the son of the son of his brother, the son of a man of the family,” the elder son said.
>
> “He was born in the family in the family and from his father, his parents. He had his father, the son of a grandfather, his father and his father. He was the father of the family. He was the father of the son of the king of the grandfather of the son of the King. He had a son of the King of the son of his father. He was a man of the father of his mother, who

The completion looks a bit like English, but is not coherent. This is expected, given that the model has not learned the dataset as well as the model trained on TinyStories. With a

larger and more diverse dataset, the model would need to be trained for longer to learn the distribution well. Additionally, the OpenWebText dataset is much noisier than TinyStories, which hinders learning, and further increases the FLOPs that would be required to train a usable model on it.

Problem (`leaderboard`): Leaderboard (6 points)

a) Final validation loss: 3.26813

Model:

I found I could get the fastest improvement in loss by severely undertraining a large model, and preferencing a short model (few layers relative to d_{model} and d_{ff}). I also used weight tying, but didn't change much else about the model (kept RoPE, SwiGLU, pre-norm with RMSNorm, MHA, etc.) Final configuration:

- $d_{\text{model}} = 1280$
- $d_{\text{ff}} = 3456$
- $\text{num\_layers} = 12$
- $\text{num\_heads} = 16$
- $\text{context\_length} = 512$
- Tied embeddings and LM head

Training code:

- `torch.compile`
- AMP with `torch.autocast`
- Pinned memory for data loading

Training parameters:

I extensively tuned the learning rate and batch size. I tested cosine scheduling, linear scheduling, and some more creative ideas (example: linear warmup, fast exponential decay, then gradual cosine/linear decay). Ultimatley a simple linear decay schedule worked best, with a mininum learning rate that was almost a third of the maximum learning rate (on the intuition that the model would be severely undertrained, and decaying the learning rate early wouldn't help).

I experimented with weight decay and optimizer betas, but observed very little difference.

- Learning rate: 1% linear warmup to `5e-4`, then linear decay to `1.8e-4`
- Batch size: 128
- Steps: 13,000
- Total tokens: ~851M

WandB Report°

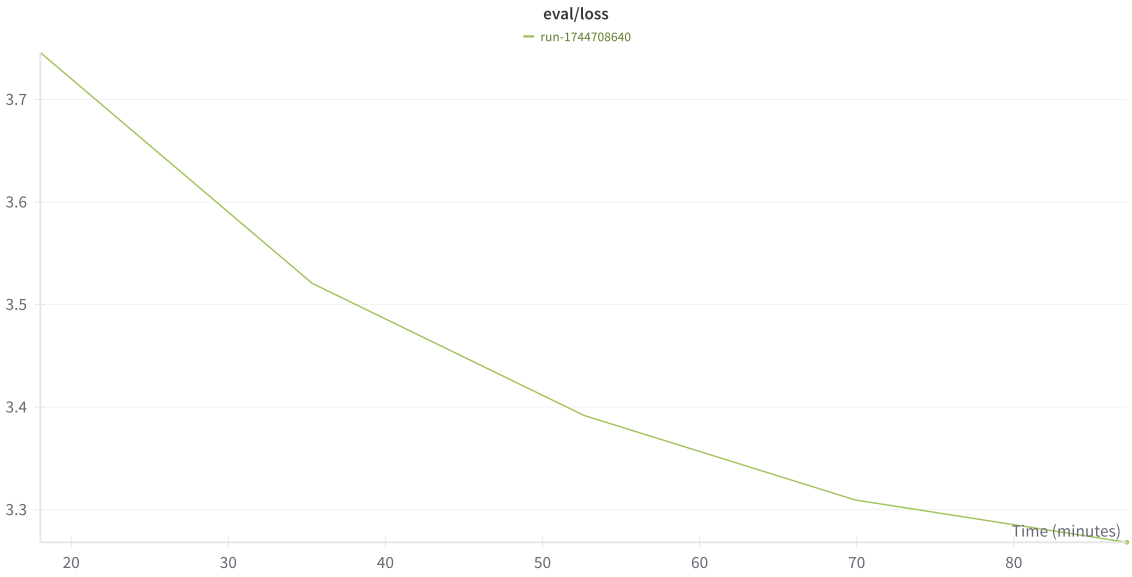

Figure 8: Leaderboard Run — Eval Loss

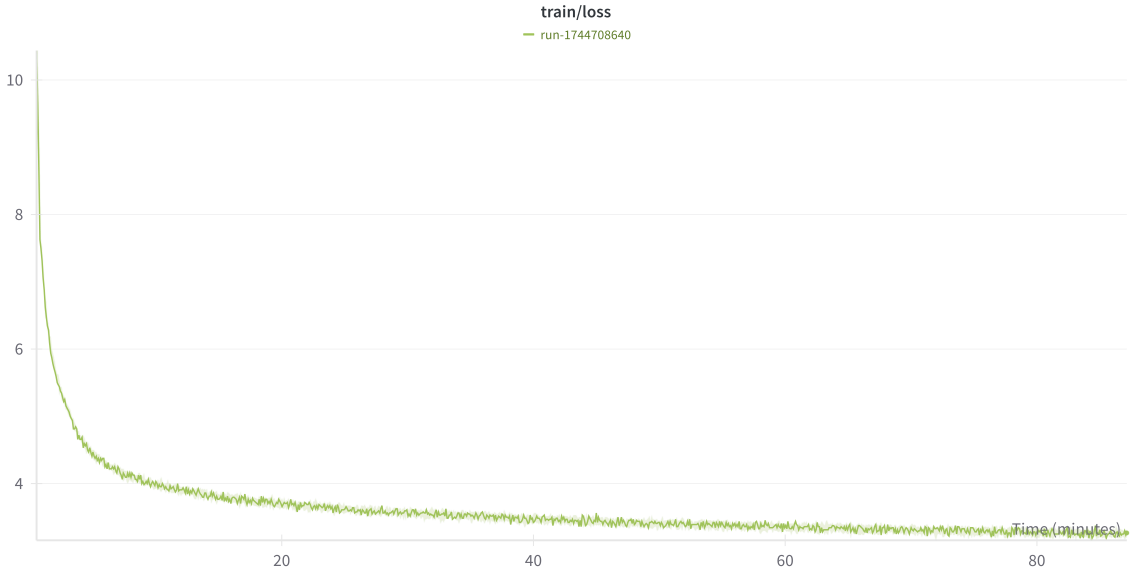

Figure 9: Leaderboard Run — Train Loss

Index of Figures