CS336 Assignment 1 Writeup

Marcel Rød
Stanford University
roed@stanford.edu

This was a huge assignment and took a lot of time. I triple-checked it, but might have missed something still – please Slack me if something is missing.

2. Byte-Pair Encoding (BPE) Tokenizer

2.1. Understanding Unicode

(a) `chr(0)` returns the byte `b'\x00'` in Python which represents the Unicode null character.

(b) When printing the string representation with `__repr__`, the null character string is shown as a `'\x00`, which is the escape sequence for the null character. If we instead use `__str__` (or just print normally), we see nothing, as the null character is not printable, and shows up as a zero-width space.

(c) Adding the character in the middle of a Python string, it will not be displayed in the output, unless the string is printed using `__repr__`. In order to see the null character shown as its escape sequence, we can use the `__repr__` method. If this were C, the string would end early at the null byte.

2.2. Unicode Encodings

(a) For UTF-8, each character is encoded using 1 to 4 bytes, but each byte has only 7 bits of data with one extra bit indicating whether or not the current character has more bytes. If we construct our tokenizer using BPE on UTF-8 encodings, we expect UTF-8 characters with more than one byte that are frequently used to be merged into a single token, but we don't necessarily pay the extra cost of explicitly constructing new tokens for lesser used characters, which is the case for UTF-16/32. Since we need to have a token or set of tokens for every character, UTF-32 would require that every Unicode character has its own token to be compatible on all text, or we need unknown token, which makes our tokenizer non-invertible.

(b) The issue here is that the function splits the text into bytes before encoding it into UTF-8. This means that the `.decode('utf-8')` call can get bytes from the middle of a UTF-8 codepoint, or codepoints that are not terminated. For instance, with the character `U+03A9` (Ω), the UTF-8 encoding is `b'\xce\xa9'`. If we split this into bytes, we get `b'\xce'` and `b'\xa9'`. `b'\xce'` is not a valid UTF-8 encoding, since the decoder expects at least one more byte to follow it. `b'\xa9'` is an invalid start byte in UTF-8, and also fails to decode.

(c) A completely invalid two-byte sequence is `b'\xF1\x80'`, which is not a valid UTF-8 encoding. Both bytes are valid, and valid in sequence, but the bytes do not form a valid UTF-8 codepoint because the decoder will continue to expect more bytes to follow the `b'\xF1'` byte.

2.4. BPE Tokenizer Training

2.4.1. (`train_bpe`)

While I started implementing this in Python, I didn't want to wait several hours to train the tokenizer on the large datasets. So I instead implemented the training in Rust with PyO3 and then called my Rust functions from Python. My code is fully parallelized, and almost as fast as the HuggingFace implementation (I need to iron out some threading contention issues)! See the `rustsrc` dir in my submission for the Rust code. My code passes the tests in `test_train_bpe.py`.

2.5. Experimenting with BPE Tokenizer Training

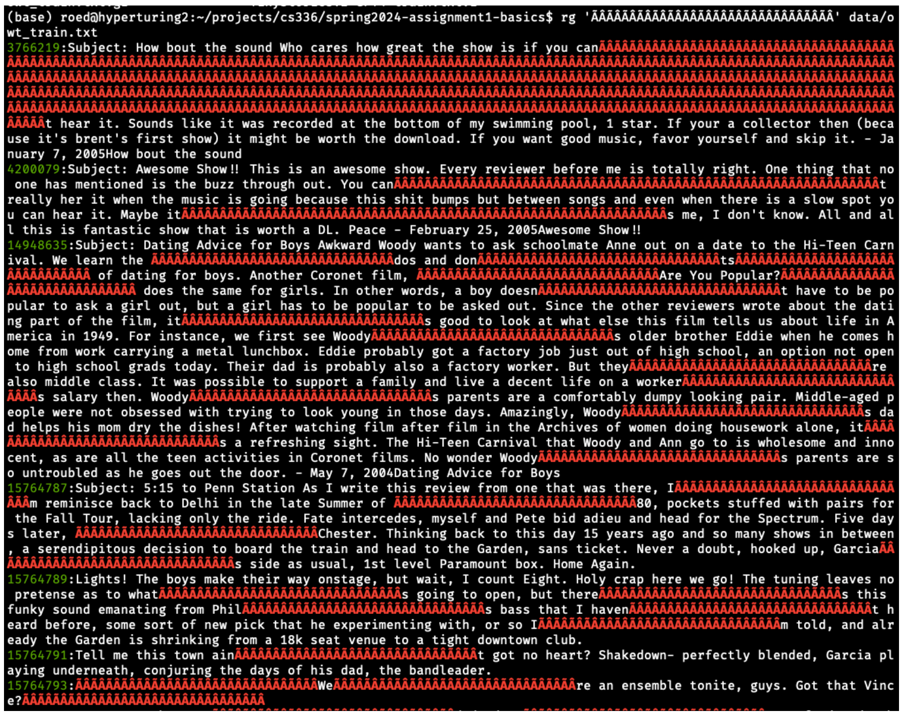
2.5.1. (`train_bpe_tinystories`)

(a) My training code takes 8.7 seconds to train with a max vocabulary size of 10,000 on the TinyStories dataset, and caps out at 19.32GB of memory usage (according to `/usr/bin/time -v <cmd>`). Note that this is excluding the time to load the dataset, but includes time to switch between Python and Rust, the time to run the initial regex and the time to create Python objects to hold the results. Running the merging process single-threaded takes around 15 seconds. The longest word in the vocabulary is `"accomplishment"`, which makes a lot of sense.

(b) Profiling my code shows that the merging process (3s) and compiling the Regex (2s) take up the most time time to switch. Since my CPU usage is at 100% during the merging process, I think that would be the next thing to optimize (algorithmically). Previously, I had kept track of the dependencies in a dictionary that went from pairs to the words they were in, but updating this dictionary caused too much contention when running multithreaded, so while this theoretically lead to fewer accesses, it parallelized significantly worse. Interestingly, the merging process takes significantly less time on my local machine (M1 Pro ARM chip), completing in less than 1 second, but the regex takes significantly longer to complete on fewer cores.

2.5.2. (`train_bpe_expts_owt`)

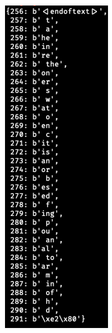
(a) On the OpenWebText dataset, the training process completes in 5 minutes and 24 seconds. The longest token in the vocabulary is a set of dashes, the token ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ, which is 32 characters long. This seems like nonsense, but looking through the dataset, it seems like a very common encoding error. Specifically, some apostrophes seem to have been encoded as very long sequences of these characters.



```
(base) roed@hyperturing2:~/projects/cs336/spring2024-assignment1-basics$ rg 'ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ' data/o
wt_train.txt
3766219:Subject: How bout the sound Who cares how great the show is if you canÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ
ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ
ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂÃÂt hear it. Sounds like it was recorded at the bottom of my swimming pool, 1 star. If your a collector then (beca
use it's brent's first show) it might be worth the download. If you want good music, favor yourself and skip it. - Ja
nuary 7, 2005How bout the sound
4200079:Subject: Awesome Show!! This is an awesome show. Every reviewer before me is totally right. One thing that no
 one has mentioned is the buzz through out. You canÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂt
really her it when the music is going because this shit bumps but between songs and even when there is a slow spot yo
u can hear it. Maybe itÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs me, I don't know. All and al
l this is fantastic show that is worth a DL. Peace - February 25, 2005Awesome Show!!
14948635:Subject: Dating Advice for Boys Awkward Woody wants to ask schoolmate Anne out on a date to the Hi-Teen Carn
ival. We learn the ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂdos and donÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂtsÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂÃÂÃÂÃÂÃÂ of dating for boys. Another Coronet film, ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂAre You Popular?ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ does the same for girls. In other words, a boy doesnÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂt have to be po
pular to ask a girl out, but a girl has to be popular to be asked out. Since the other reviewers wrote about the dati
ng part of the film, itÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs good to look at what else this film tells us about life in A
merica in 1949. For instance, we first see WoodyÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs older brother Eddie when he comes h
ome from work carrying a metal lunchbox. Eddie probably got a factory job just out of high school, an option not open
 to high school grads today. Their dad is probably also a factory worker. But theyÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂre
also middle class. It was possible to support a family and live a decent life on a workerÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ
ÃÂÃÂs salary then. WoodyÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs parents are a comfortably dumpy looking pair. Middle-aged p
eople were not obsessed with trying to look young in those days. Amazingly, WoodyÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs da
d helps his mom dry the dishes! After watching film after film in the Archives of women doing housework alone, itÃÂÃÂ
ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs a refreshing sight. The Hi-Teen Carnival that Woody and Ann go to is wholesome and inno
cent, as are all the teen activities in Coronet films. No wonder WoodyÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs parents are s
o untroubled as he goes out the door. - May 7, 2004Dating Advice for Boys
15764787:Subject: 5:15 to Penn Station As I write this review from one that was there, IÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃ
ÂÃÂm reminisce back to Delhi in the late Summer of ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ80, pockets stuffed with pairs for
 the Fall Tour, lacking only the ride. Fate intercedes, myself and Pete bid adieu and head for the Spectrum. Five day
s later, ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂChester. Thinking back to this day 15 years ago and so many shows in between
, a serendipitous decision to board the train and head to the Garden, sans ticket. Never a doubt, hooked up, GarciaÃÂ
ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs side as usual, 1st level Paramount box. Home Again.
15764789:Lights! The boys make their way onstage, but wait, I count Eight. Holy crap here we go! The tuning leaves no
 pretense as to whatÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs going to open, but thereÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs this
funky sound emanating from PhilÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂs bass that I havenÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂt h
eard before, some sort of new pick that he experimenting with, or so IÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂm told, and alr
eady the Garden is shrinking from a 18k seat venue to a tight downtown club.
15764791:Tell me this town ainÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂt got no heart? Shakedown- perfectly blended, Garcia pl
aying underneath, conjuring the days of his dad, the bandleader.
15764793:ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂWeÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂre an ensemble tonite, guys. Got that Vinc
e?ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ
```

Figure 1: The dataset contains a lot of these encoding errors, and some of them grow incredibly long.

(b) From reading through the vocabularies of both tokenizers, noting the order in which the tokens were added, it seems like the tokenizer trained on TinyStories is quite proper, and as expected, while the tokenizer trained on OpenWebText has a lot of strange tokens related to a lack of character standardization as well as some encoding errors.



```
{256: b'◁endoftext▷',
 257: b' t',
 258: b' a',
 259: b'he',
 260: b'in',
 261: b're',
 262: b' the',
 263: b'on',
 264: b'er',
 265: b' s',
 266: b' w',
 267: b'at',
 268: b' o',
 269: b'en',
 270: b' c',
 271: b'it',
 272: b'is',
 273: b'an',
 274: b'or',
 275: b' b',
 276: b'es',
 277: b'ed',
 278: b' f',
 279: b'ing',
 280: b' p',
 281: b'ou',
 282: b' an',
 283: b'al',
 284: b' to',
 285: b'ar',
 286: b' m',
 287: b' in',
 288: b' of',
 289: b' h',
 290: b' d',
 291: b'\xe2\x80'}
```

Figure 2: As early as the 36th merge for the OWT tokenizer, we see a Unicode code point with more than one byte. This is the start of the token `\xe2\x80\x99`, which is the UTF-8 encoding for the right single quotation mark (').

2.6. BPE tokenizer: Encoding and Decoding

2.6.1. (tokenizer_experiments)

(a) Storing tokens as u16, the compression ratio (bytes → token bytes) is 2.06 for TinyStories, and 2.35 for OpenWebText. For bytes → tokens, the compression ratio is 4.12 and 4.70 respectively. This difference is expected, since the OpenWebText tokenizer was allowed a larger vocabulary.

(b) Crossing the tokenizers gives us a compression ratio of 2.01 (4.02 bytes → tokens) for TinyStories using the OWT tokenizer, and 1.60 (3.20 bytes → tokens) for OpenWebText using the TinyStories tokenizer. This is expected, since the tokenizers are trained on different datasets, and the OpenWebText tokenizer has a lot of strange tokens that are not present in the TinyStories tokenizer.

(c) My tokenizer encodes at 466 MB/s with the TinyStories tokenizer, and at 297 MB/s with the OpenWebText tokenizer.

At this speed it would take roughly 29 minutes and 30 seconds to tokenize the entirety of The Pile.

(d) `uint16` is an appropriate choice because it allows for integers up to 65535, which is more than enough for the vocabulary size of 32,000 used for OpenWebText. The `uint32` type would be overkill, and the `uint8` type would not be able to hold the vocabulary size.

3. Transformer

3.5. The Full Transformer LM

3.5.1. (transformer_accounting)

(a) We have one token embedding matrix, a positional embedding matrix, a language head matrix, a final RMSNorm, and num_layers sets of blocks, where each block has two layer norms, a self-attention layer and a feed-forward layer. Each self-attention layer has a projection layer for each of Q, K and V, as well as an output transformation matrix. Each feed-forward layer has a two linear layers.

$$\underbrace{\text{vocab\_size} \cdot \text{d\_model}}_{\text{token embedding}} + \underbrace{\text{context\_length} \cdot \text{d\_model}}_{\text{positional embedding}} + \underbrace{\text{d\_model}}_{\text{RMSNorm}} + \underbrace{\text{d\_model} \cdot \text{vocab\_size}}_{\text{LM Head Matrix}}$$
$$+ \text{num\_layers} \cdot \underbrace{\left(\underbrace{2 \cdot \text{d\_model}}_{\text{RMSNorm}} + \underbrace{3 \cdot \text{d\_model}^2 + \text{d\_model}^2}_{\text{Self Attention}} + \underbrace{2 \cdot \text{d\_model} \cdot \text{d\_ff}}_{\text{Feed Forward}} \right)}_{\text{Transformer Block}}$$

One could also consider the case where the embeddings and the LM head matrix are tied (they share weights), but I will not do so. This gives a total parameter count of 1,637,176,000. For single-precision floating point numbers, this is 6,548,704,000 of memory, or 6.55 GB.

For a matrix multiplication of two matrices of size $(n \times o)$ and $(o \times m)$ (producing output $(n \times m)$), the number of FLOPs is $2nmo$. Let's enumerate the matrix multiplications and their FLOPs

Classification	Count	Shapes	Total FLOPs
Attention Projections	4 num_layers	(context_length · d_model), (d_model · d_model)	8 num_layers context_length d_model^2
Attention	2 num_layers	(context_length · d_model), (d_model · context_length)	4 num_layers context_length^2 d_model
Feed-Forward MM	2 num_layers	(context_length · d_model), (d_model · d_ff)	4 num_layers context_length d_model d_ff
LM-head	1	(context_length · d_model), (d_model, vocab_size)	2 context_length d_model vocab_size

For GPT-2 XL, we get the following:

Classification	Total FLOPs	Percentage
Attention Projections	1,006,632,960,000	28.7%
Attention	322,122,547,200	9.18%
Feed-Forward MM	2,013,265,920,000	57.4%
LM-head	164,682,137,600	4.70%
Total	3,506,703,564,800	

FLOPs for GPT-2 XL

The total number of FLOPs is 3.5T for a single forward pass at batch size 1.

(b) For GPT-2 XL, the feed-forward layers take up the majority of the model FLOPs. After this, the attention projections are also quite significant. The attention mechanism itself is only 9% of the total FLOPs, which is quite surprising.

(c)

Classification	Total FLOPs	Percentage
Attention Projections	57,982,058,496	19.9%
Attention	38,654,705,664	13.3%
Feed-Forward MM	115,964,116,992	39.8%
LM-head	79,047,426,048	27.1%
Total	291,648,307,200	

FLOPs for GPT-2 Small

Classification	Total FLOPs	Percentage
Attention Projections	206,158,430,208	24.9%

Classification	Total FLOPs	Percentage
Attention	103,079,215,104	12.5%
Feed-Forward MM	412,316,860,416	49.9%
LM-head	105,396,568,064	12.7%
Total	826,951,073,792	

FLOPs for GPT-2 Medium

Classification	Total FLOPs	Percentage
Attention Projections	483,183,820,800	27.2%
Attention	193,273,528,320	10.9%
Feed-Forward MM	966,367,641,600	54.5%
LM-head	131,745,710,080	7.42%
Total	1,774,570,700,800	

FLOPs for GPT-2 Large

Varying the model proportions seems to make the attention mechanism proportionally more dominant. Additionally, smaller models can't scale down the vocab size, so the LM-head FLOPs are more significant for smaller models. Because of this effect it's harder to see the attention becoming proportionally larger than the feed-forward layers with decreasing model size.

(d) We increase the context length to 16,384 for GPT-2 XL:

Classification	Total FLOPs	Percentage
Attention Projections	16,106,127,360,000	12.1%
Attention	82,463,372,083,200	61.8%
Feed-Forward MM	32,212,254,720,000	24.1%
LM-head	2,634,914,201,600	1.9%
Total	133,416,668,364,800	

FLOPs for GPT-2 XL with context length 16,384

Clearly the proportion of FLOPs required for the attention mechanism increases significantly with the context length. This is because the attention mechanism scales quadratically with the context length, while the feed-forward layers scale linearly.

4. Training a Transformer LM

4.2. The SGD Optimizer

4.2.1. `(learning_rate_tuning)`

When running at 1e1, the loss goes down to 3.7 after 10 epochs. When running at 1e2, it goes down to 1.67e-23. At 1e3 it diverges to 1.91e18 after 10 epochs, and adding more epochs has it diverge to `NaN`. Clearly 1e1 is too low a learning rate, and 1e3 is too high, but 1e2 seems to be a good learning rate for this model and optimizer.

4.3. AdamW

4.3.1. (`adamwAccounting`)

(a) Let's go through all the parts that require memory: We have our existing expression for the parameters, as seen above. Collapsing all the terms with $\text{d\_ff} = 4\ \text{d\_model}$, we get

$$\text{n\_parameters} = \text{d\_model} \times (\text{num\_layer} \times (2 + 12\ \text{d\_model}) + 2\ \text{vocab\_size} + \text{context\_length} + 1)$$

For activations we only need to store the maximum length of all the activations in the model, which will be the maximum of either the Q, K and V values when stored together or the intermediate activations in the feed-forward layer. The expression turns out to be:

$$\text{n\_activations} = \text{batch\_size} \times \text{context\_length} \times \max(3 \times \text{d\_model}, \text{d\_model} + \text{context\_length}, \text{d\_ff})$$

where the first option is the activations for Q, K and V together, the second is the activations for the attention weights together with the V activations, and the third is the intermediate activations in the feed-forward. Note that we can optimize the memory requirement for the attention mechanism using memory efficient attention, either from `xformers`, or from FlashAttention. Both of these approaches are now supported natively in PyTorch.

For gradients, the memory requirement is the same as for parameters.

AdamW stores two sets of state tensors for each parameter in the model, one for the momentum and another for the second moment.

Assuming $\text{d\_ff} = 4 \times \text{d\_model}$, the final tally is

$$\begin{gathered} 4\ \text{bytes} \times 4 \times \text{n\_parameters} + \text{n\_activations} \\ = 4\ \text{bytes} \times 4 \times \text{d\_model} \times (\text{num\_layer} \times (2 + 12\ \text{d\_model}) + 2\ \text{vocab\_size} + \text{context\_length} + 1) \\ +4\ \text{bytes} \times \text{batch\_size} \times \text{context\_length} \times \max(3 \times \text{d\_model}, \text{d\_model} + \text{context\_length}, \text{d\_ff}) \end{gathered}$$

(b) Computing the coefficients, we get

$$26{,}214{,}400 \cdot \text{batch\_size} + 26{,}194{,}816{,}000.$$

Solving for the batch size with a total budget of 80GB, we theoretically can hold a batch size of 2052.5, so around 2048. In practice we have to consider intermediary buffers and overheads, so it's unlikely we can actually reach this batch size.

(c) One step of AdamW (after computing gradients) takes a set number of operations per parameter in the model. It takes 3 operations to update the first moment estimate, 4 operations to update the second order moment estimate, 5 operations to compute the parameter updates, and 2 operations to apply weight decay. In total then, we have $14 \times \text{n\_parameters}$ or

$$14 \times \text{d\_model} \times (\text{num\_layer} \times (2 + 12\ \text{d\_model}) + 2\ \text{vocab\_size} + \text{context\_length} + 1)$$

FLOPs per step.

(d) We can reference our numbers from earlier. A single step will take $3 \cdot \text{batch\_size} \cdot 3{,}506{,}703{,}564{,}800$ FLOPs to do the forward and backward passes, and $14 \cdot 1{,}637{,}176{,}000$ FLOPs to do the optimizer step (negligible). That totals 10,772,616,271,529,600 FLOPs per step, or 10.77 PFLOPs! If we have an MFU of 50%, we can do $0.5 \cdot 19.5 = 9.75$ TFLOP/s. Then 400K steps with batch_size 1024 would take

$$\frac{10.77 \cdot 10^{15} \cdot 400 \cdot 10^{3}}{9.75 \cdot 10^{12}} = 441846153.4\ \text{s} = 5{,}113.96\ \text{days}$$

Note that this is unrealistic, since in practice one would use the Tensor Cores and TF32 instead of FP32, which are much much faster on the A100 (this gap is even higher on H100). This would be 8 times faster for single-precision floats (156 TFLOPS), and you could instead get a time of 639.25 days.

5. Training Loop

5.3. Training loop

5.3.1. `(training_together)`

The script has been added in `cs336_basics/training_loop.py`. It does much more than required, but certainly uses `np.memmap`, checkpoint serialization for later use, periodic logging, and manages hyperparameters from the command line. The script is kitted out with a ton of features after optimizing for the leaderboard submission in the final task, particularly ways to control the model architecture and training dynamics.

6. Generating Text

6.1.1. `(decoding)`

The function works as expected, and generates text until it hits "`<|endoftext|>`". In my initial run of the function, I noticed that the "`<|endoftext|>`" token was being added in smaller subtokens, namely "`<|`", "`endoftext`", "`|>`". This was because when I had run my decoding process, I forgot to specify that "`<|endoftext|>`" should be treated as a special token, so the pretokenizer split it into three separate tokens. All the recommended features are supported.

7. Experiments

7.1.1. `(experiment_log)`

I use Weights & Biases to log my experiments, and have included plots and descriptions of the experiments I did in the following tasks.

7.1.2. `(learning_rate)`

(a) I tried the learning rates seen in the plot below, forming a geomspace sweep from 1e-5 to 1e1 with 7 steps. The 1e-3 run has a final validation loss of 1.246, which passes the stated requirement of 1.45. Note that the training losses are smoothed by being averaged, and so they aren't quite representative. I discovered that the training losses were somewhat shifted very late in experimentation, but luckily there were no issues with the validation losses. The learning rate of 1e-3 is the highest learning rate that does not diverge, and leads to the lowest loss. 1e-2 diverges very slowly, but 1e-1 diverges after a few steps, as can be seen by the blue dots in the plot denoting NaN values.

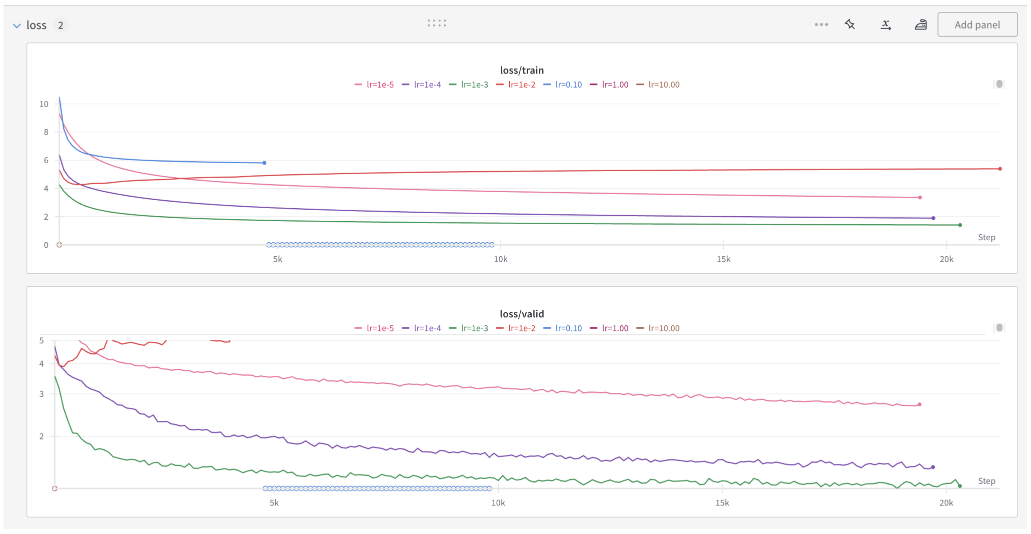


Figure 3: The results for different learning rates.

(b) Note that the best result comes from the 1e-3 learning rate, which is the highest value that did not diverge. This seems to corroborate the idea that the learning rate should be as high as possible without diverging, as the model learns faster with a higher learning rate. Toward the end of optimization this is not true, however, as we want the model to settle to a local minimum once we've trained for a bit at higher learning rates.

7.1.3. `(batch_size_experiment)`

(a) I tried batch sizes 1, 64 128, 256 and the experiment crashed with 512. It seems like the results are better for smaller batch sizes, but for the larger batch sizes we get higher throughput which means more samples per second. Thus we want a small batch size that still keeps the GPU utilization at 100%. It's often said that a low batch size is a form of regularization through noise. With a good learning rate and momentum, this can certainly be true, and the relatively higher number of total update steps per time unit can make up for the lower number of samples per update.

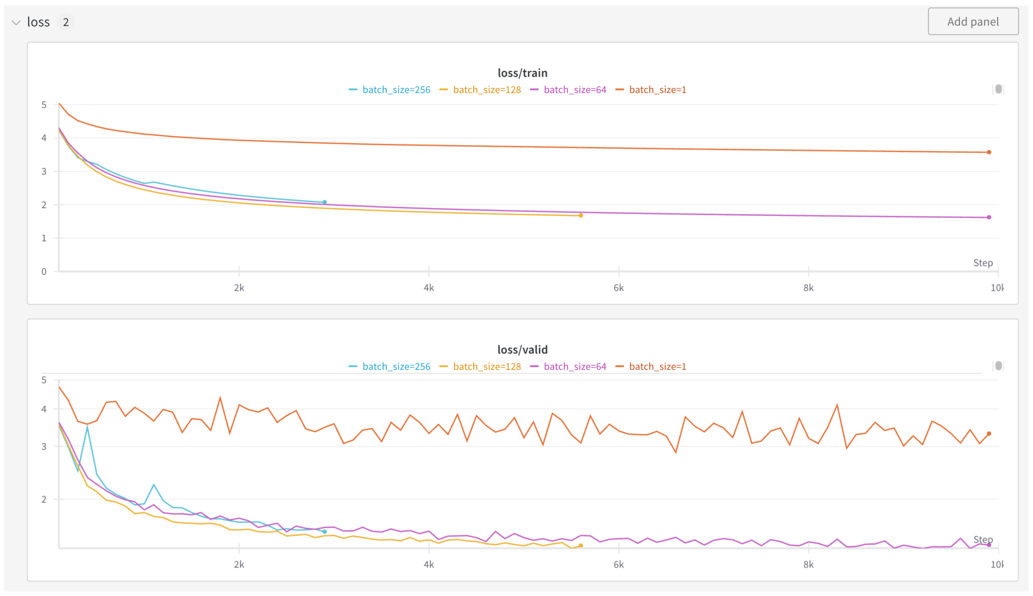


Figure 4: The results for different batch sizes.

7.1.4. `(generate)`

Here's a sample of generated text from the 128 batch size using 1e-3 learning rate on TinyStories:

```
Input text:
Her head exploded

Output text:
Her head exploded, and she dropped the blocks. She was scared and sad. She cried and called
for her mom. "Lily, what did you do? You ruined my stack of blocks!"
Her mom came and saw the mess. She was angry and worried. She hugged Lily and said, "Lily,
what are you doing? You are a bad girl. You should not have played with the blocks. You
should say sorry and clean up."
Lily felt bad and said, "I'm sorry, mom. I wanted to build a big stack. I didn't mean to make
a mess."
Her mom sighed and said, "I know you like to build, Lily. But you have to learn to share and
be kind. And you have to clean up the blocks and the blocks. And you have to ask before you
use something. Do you understand?"
Lily nodded and said, "Yes, mom. I understand. I'm sorry, mom. I will try to be a good girl.
I will help you clean up the blocks and the blocks."
Her mom smiled and said, "Thank you, Lily. I'm proud of you. I love you."
<|endoftext|>
```

The text is quite fluent and reasonably coherent. I think part of the reason why this works well is that the TinyStories dataset is clean, and has a lot of text that fits neatly into the same formats and tropes. The tokenizer is fairly simple and doesn't have to deal with many edge cases, so part of the loss being so low is that there are fewer tokens to learn and choose between. This lack of confounding factors contributes to quite decent text generation, but it likely would not generalize to more complex prompts, regressing to what it has seen in its dataset.

7.3. Ablations

7.3.1. (`parallel_layers`)

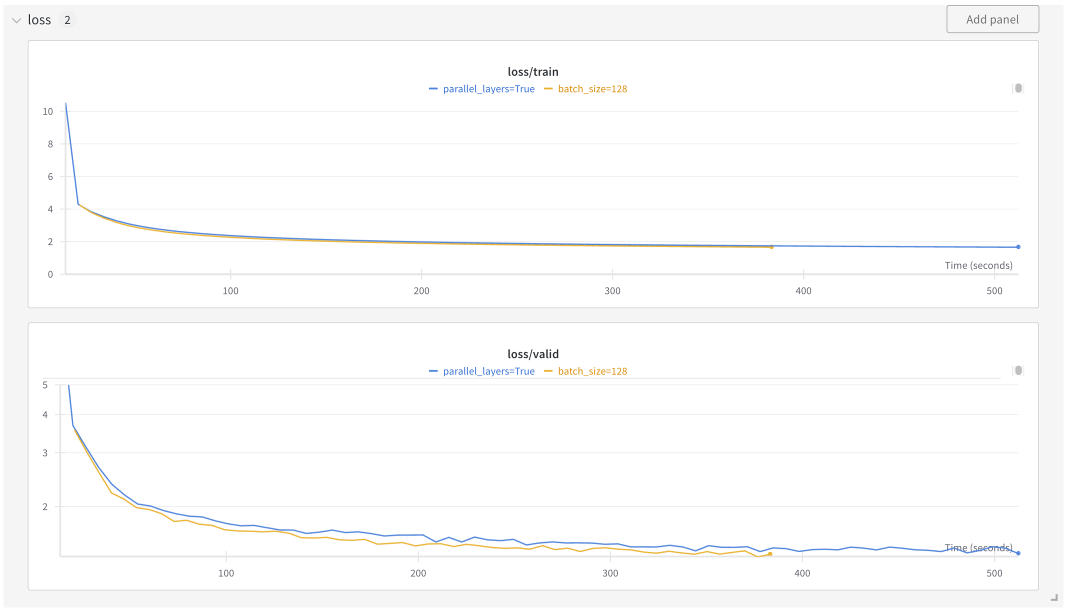


Figure 5: With and without parallel layers with the x axis being wall clock time.

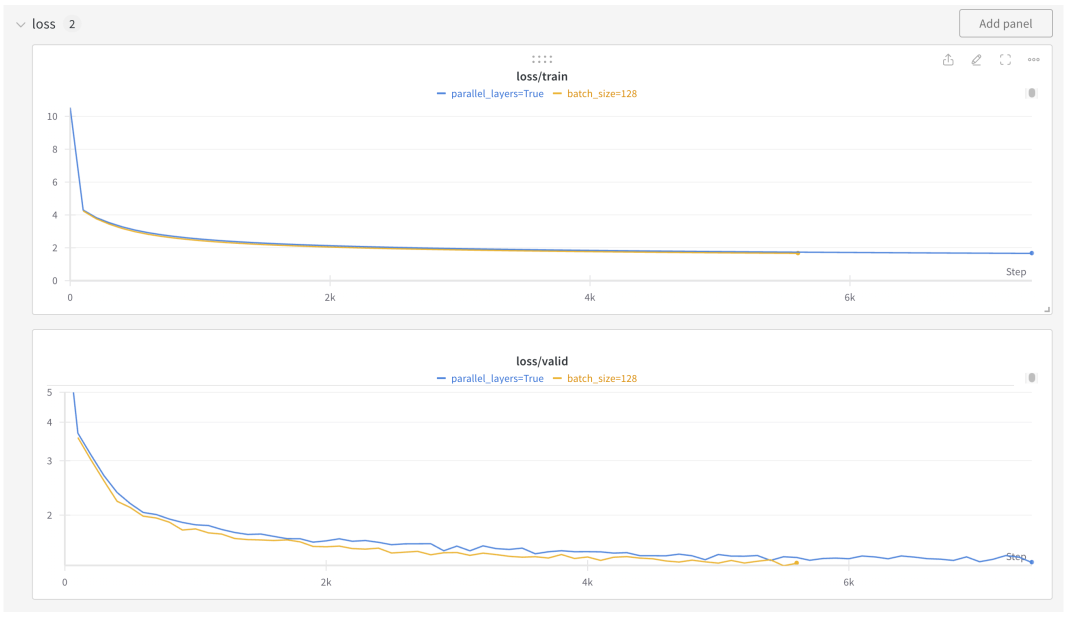


Figure 6: With and without parallel layers with the x axis being gradient steps.

After having double and triple checked this, the curves seem to be not much different, meaning the parallel layers don't seem to have much of an effect on the training time. I suspect this is because the CUDA kernels are launched asynchronously and pipelined quite well, as well as the model not being very deep and being compiled with `torch.compile`.

7.3.2. (`layer_norm_ablation`)

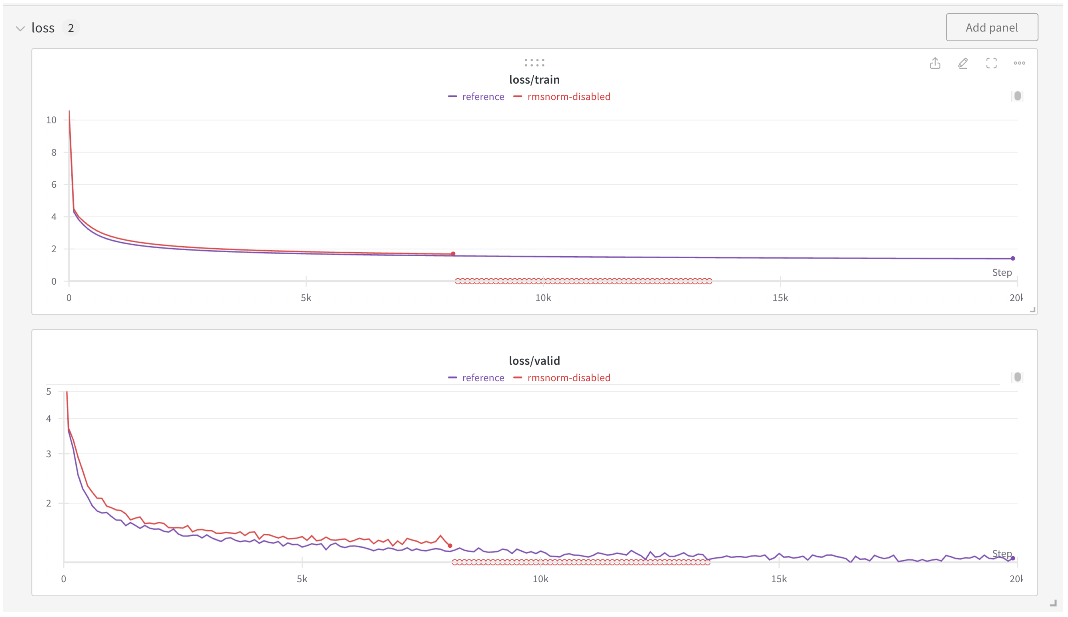


Figure 7: With and without layer normalization.

We notice that the model without the layer normalization gets destabilized and has a loss value of NaN after roughly 8k steps. This does not happen to the reference model (which I ran an extended run for to compare). The layer normalization is crucial for training stability, and ensures that the activations signals don't explode or vanish. Removing the RMSNorm seems to be fine at first, but the model quickly diverges.

7.3.3. (`pre_norm_ablation`)

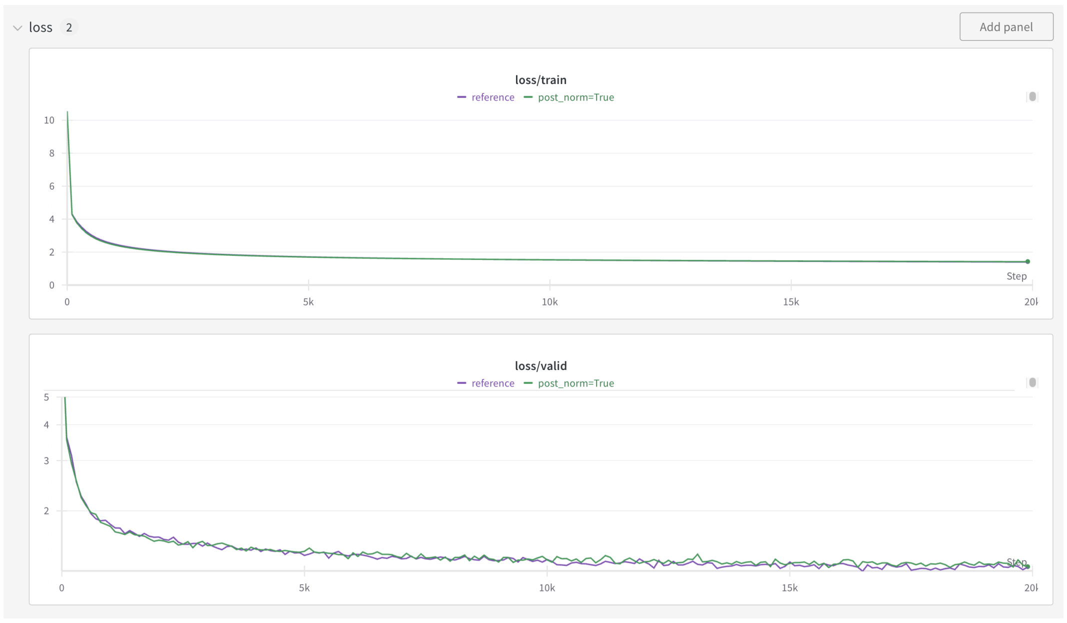


Figure 8: With and without pre-norm.

It seems like the post-norm model is not much different from the pre-norm model, but I think this is because we are not using a very deep model. With more layers, the network depends on residuals more, and the post-norm model will not retain clean residuals the way the pre-norm does. From my experience the effect of this instability is also more pronounced when training for longer, so perhaps these runs would have diverged after training for a few hours.

7.4. Running on OpenWebText

7.4.1. (`main_experiment`)

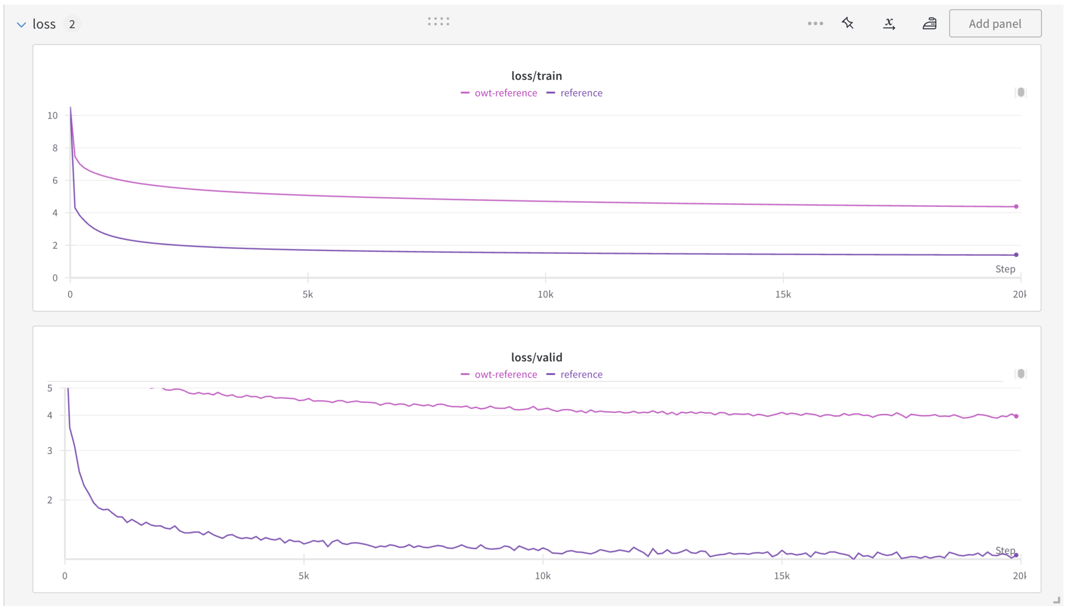


Figure 9: Training on OWT.

The OWT losses are significantly higher than the TinyStories losses, but this is expected since the OWT dataset has a much more messy distribution than TinyStories. Additionally, the OWT tokenizer has much more tokens than the TinyStories tokenizer, which means the model has to learn a much more complex distribution. These aspects contribute to the loss value being higher than that for TinyStories, which contains a very clean set of texts. The generated example for OWT follows:

```
# Input text:
Her head exploded

# Output text:
Her head exploded, and she sat her flaccid floss on the nightstand before she did. Her
posture not even touches her chest, but her eyes narrowed, moving slowly around, and then
starting to swing out of her posture, causing Her ability to instantly decipher the cell’s
heart and move her head upward, or to smite her neck while her hips were putting her back on
the block. Her hand touched her chest as it was that many wanted to look at her and
everything was there, but she never moved it so quickly, and there was no searching. In fact,
it was the odd look she was exposed to and took her breath, and now the fire started to go
off.

The overheating clothes and morsel took longer. Her was still drawn to this unpleasant
experience, but her thoughts and emotions had come back from her terrorizing own thoughts.
Her thoughts dropped, and the feeling she had with her posture became lost.

Her sharp glance suggested that her mind was only one of two things. She felt like a dovet
covering her skin and whether it felt great for her, or for herself, she felt she should feel
more comfortable with, or if she felt more normal. Her stamina was not the only reason her
body would be, she had to concentrate and cheer her pain down on her head. She felt like I
had a lesson for her. She hadn’t been able to take her breath and breath, but now that she
was feeling she did feel grateful that I was there.

Her shoulders reached the receptive threshold, and Her skin returned to that higher level,
only having to sit back in her chair. Her body began to vibrate, and both feet pressed
forward, wide to the length of her head as he shintered through the inward wall. Her scalpel
now exposed slightly, his outer thigh full of soft muscles, and her shoulders were active,
the less as if to strain her muscles and comfort her composure. Her viscera had changed
slightly as she felt around her, but as soon as she felt the pain felt great, she felt as
strong as she could. Her brushstroke was very sharp, and it immediately faded, though Her
skull of her forearm. Her long glaze accelerated and formed his focus on vision. It felt like
a whole new day, and her chest held no stand or push from any human contact.

Her eyes blurred in her gaze, but she felt the result been impressive. Her ears and
```

This text is not completely incoherent, but it is certainly strange to read. It seems to do more with the prompt, referring to “Her” as well as “head” several times in the text. This is pretty OK for something trained for such a short time on such a crazy dataset, but nothing to write home about. The reason why this text isn’t as coherent as the TinyStories text is likely because the OWT dataset is much more varied, less structured and has a lot more noise in it along with the tokenizer being more complex.

7.5. Leaderboard

My final loss (as of writing this) was 3.12 after 82 minutes, which puts me at #1 on the leaderboard! My key tricks were to compile and speed up the model in every possible way (using mixed precision in PyTorch), rotary embeddings, learning rate tuning, and scaling up the model size. I use 12 layers, 16 heads, d_ff of 4096, and a d_model of 1024. I employ a gated MLP mechanism inspired by Mistral AI, tie my embeddings to the LM head and use the SiLU activation function. Importantly, I’m also very careful to initialize the weights to reasonable values. With tied weights, it’s important to initialize them to something that works for matrix multiplication, e.g. with the `kaiming_uniform_` function in PyTorch. Ablating that one factor leads to a horrible loss value.

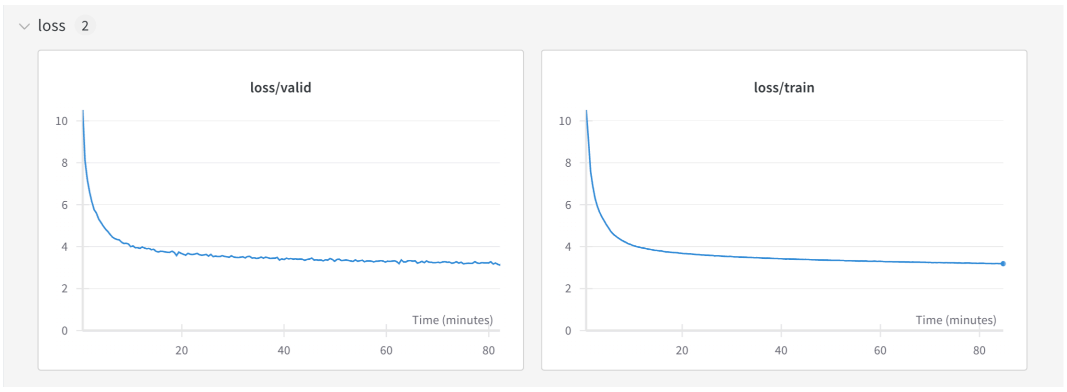


Figure 10: The final training curve.