CS336 Assignment 1

Dmitry Frumkin

November 2, 2025

Contents

2 Byte-Pair Encoding

2.1 The Unicode Standard

(a) **What Unicode character does chr(0) return?**

It returns the unicode character whose integer code is 0, aka the null character.

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

The character's string representation is `'\x00'`; however, it's printed representation is empty, i.e. nothing is printed.

(c) **What happens when this character occurs in text?**

In the given example, the string representation is `'this is a test\x00string'`, but `this is a teststring` gets printed.

2.2 Unicode Encodings

(a) **What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32?**

Most of the Internet is in UTF-8 because it results in denser encodings (English and other common characters use only 8 bytes). By training on UTF-8 we get shorter sequences and also avoid having to recode the inputs into UTF-16/32.

(b) **Consider the following (incorrect) function, which is intended to decode a UTF-8 byte string into a Unicode string. Why is this function incorrect? Provide an example of an input byte string that yields incorrect results.**

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in bytestring])

>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
```

For the Japanese example that you provided (" こんにちは"), we will get `UnicodeDecodeError`. It does not make sense to decode individual bytes because most Japanese characters are represented by 3 bytes in UTF-8 (here, we get 15 bytes for 5 characters).

(c) **Give a two byte sequence that does not decode to any Unicode character(s).**

```
>> b'\xC2\x00'.decode('utf-8')
UnicodeDecodeError: <...>
```

The first byte 0xC2 indicates that it's a two-byte sequence; however, the second byte must be in a certain range (0x80–0xBF), and here it's not.

2.5 Experimenting with BPE Tokenizer Training

- **Code:** `cs336_basics/train_bpe.py`, `cs336_basics/pretokenizer.py`
- **Reports:** `bpe_train_reports/`

I ran BPE training on a MacBook Air M4 (10 cores) with 32 GB memory, which is probably inferior to the dedicated resource that Stanford students used. Profiling with Scalene may have added a small overhead.

BPE Training on TinyStories

(a) **How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?**

Training took 1m37s with peak memory usage of 320 MB as reported by scalene with 613 MB as the sum of peaks (theoretical overestimation). The longest tokens in the vocabulary are '*accomplishment*', '*disappointment*', '*responsibility*' - common English words as expected for a dataset consisting of English stories; their encodings are 15 bytes long, which makes sense since we start with 256 single bytes and one 13-byte-long special token and perform 10,000 merges.

(b) **Profile your code. What part of the tokenizer training process takes the most time?**

Pre-tokenization took 40s, of which parsing with a regular expression took 33s, and merging took 55s.

BPE Training on OpenWebText

(a) **How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?**

Training took 10h45m with peak memory usage of 673 MB as reported by scalene with 7.8 GB as the sum of peaks (theoretical overestimation, though htop showed an increase of memory usage closer to this number); nearly all the time (10h15m) went on token merging because we have a much larger and diverse input and perform more merges. The longest tokens are '*ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ*' (an artifact from erroneous double UTF-8 encoding) and '--' (a common text separator); their encodings are 64 bytes long, which is reasonable since we perform 32,000 merges.

(b) **Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.**

Both tokenizers mostly work with English text; however, OpenWebText tokenizer was trained on a much more diverse dataset (reflected in a larger vocabulary size), and it also deals with various non-text symbols and even corrupted sequences (as shown above) that were not filtered out.

2.6 Implementing the tokenizer

Code: `cs336_basics/tokenizer.py`

2.7 Experiments

Code: `cs336_basics/sampler.py`, `cs336_basics/tokenizer.py`

(a) **Sample 10 documents from TinyStories and OpenWebText. Using your previously-trained TinyStories and OpenWebText tokenizers (10K and 32K vocabulary size, respectively), encode these sampled documents into integer IDs. What is each tokenizer' s compression ratio (bytes/token)?**

The compression ratio is 4 bytes/token for TinyStories and 4.4 bytes/token for OpenWebText. It is higher for TinyStories because its vocabulary is much larger: 32,000 vs 10,000 for TinyStories - and it overcompensates for the higher entropy of the OpenWebText dataset.

(b) **What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens.**

For the OpenWebText sample with the TinyStories tokenizer we get $3 <$ 4.4 bytes/token and for the TinyStories sample with the OpenWebText tokenizer we get $3.8 < 4$ bytes/token. While mismatching compression schemes results in lower compression ratios; the degradation is expectedly worse when the simpler TinyStories tokenizer is used on the more complex OpenWebText dataset.

(c) **Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825 GB of text)?**

On my computer (MacBook Air, M4), the throughput of both tokenizers is about 1.5 MB/sec. Assuming the same throughput of a Pile tokenizer on the Pile dataset, it would take about 6.5 ($825 * 1024/1.5/3600/24$) days to encode. The estimate is for a single core, so we can easily divide the file (and the encoding time) by the number of available cores. We could also cache results for pretokens to significantly speed up encoding with minimal effort.

(d) **Using your TinyStories and OpenWebText tokenizers, encode the respective training and development datasets into a sequence of integer token IDs. We' ll use this later to train our language model. We recommend serializing the token IDs as a NumPy array of datatype `uint16`. Why is `uint16` an appropriate choice?**

Token indices are nonnegative integers between 0 and vocabulary size - 1. Our vocabularies' sizes are 10,000 and 32,000, which are both between $2^8 = 256$ and $2^{16} > 64,000$; therefore, it is appropriate to use a 2-byte unsigned integer datatype, that is `uint16`.

3 Transformer Language Model Architecture

- **Code:** `cs336_basics/model.py`
- **Resource Accounting:** `cs336_basics/model_analysis.ipynb`

(a) **GPT-2 XL parameters and memory** 2,127,057,600 trainable parameters and about 8.5G memory (approximately, ignoring the precomputed RoPE buffer and other overhead).

(b) **GPT-2 XL matrix multiplications and FLOPs**

- Q,K,V computation: $3 * context\_length * d_{\text{model}}^2$
- Compute and apply the score matrices: $2 * context\_length^2 * d_{\text{model}}$
- Attention output: $context\_length * d_{\text{model}}^2$
- FFN (3 matrix multiplications): $3 * context\_length * d_{\text{model}} * d_{\text{ff}}$
- LM output: $context\_length * d_{\text{model}} * vocab\_size$

In total, we perform ~4.431 TFLOPs.

(c) **GPT-2: What requires the most FLOPs**

Nearly all the FLOPs go to transformer layers with over 2/3 to the feed-forward network (SwiGLU) because of the relatively large d_{ff} and the rest to self-attention.

(d) **Repeat your analysis with GPT-2 small (12 layers, 768 d_{model}, 12 heads), GPT-2 medium (24 layers, 1024 d_{model}, 16 heads), and GPT-2 large (36 layers, 1280 d_{model}, 20 heads). As the model size increases, which parts of the Transformer LM take up proportionally more or less of the total FLOPs?**

```
======== XL ========

Transformer: 4,430,995,456,000 (48 layers: 98.1%, lm_head: 1.9%)
  Layer: 90,596,966,400
```

```
    Attention: 30.6% (qkv: 17.4%, scaled_dp_attn: 7.4%, out: 5.8%)
    FFN: 69.4%

======== large ========

Transformer: 2,554,269,532,160 (36 layers: 97.4%, lm_head: 2.6%)
  Layer: 69,122,129,920
    Attention: 27.2% (qkv: 14.6%, scaled_dp_attn: 7.8%, out: 4.9%)
    FFN: 72.8%

======== medium ========

Transformer: 1,328,303,570,944 (24 layers: 96.0%, lm_head: 4.0%)
  Layer: 53,150,220,288
    Attention: 24.2% (qkv: 12.1%, scaled_dp_attn: 8.1%, out: 4.0%)
    FFN: 75.8%

======== small ========

Transformer: 498,548,342,784 (12 layers: 92.1%, lm_head: 7.9%)
  Layer: 38,252,052,480
    Attention: 21.1% (qkv: 9.5%, scaled_dp_attn: 8.4%, out: 3.2%)
    FFN: 78.9%
```

As the model size increases, the fraction of FLOPs that goes to attention increases, specifically the part where we compute queries, keys and values.

(e) **Take GPT-2 XL and increase the context length to 16,384. How does the total FLOPs for one forward pass change? How do the relative contribution of FLOPs of the model components change?**

The number of FLOPs jumps from ~4.4 TFLOPs to ~148.2 TFLOPs. With the larger context length, now over 2/3 of a single block's FLOPs go to attention with the computation and application of score matrices (proportional to $context\_length^2$) taking 56% of the block's FLOPs.

4 Training a Transformer LM

4.1 Cross-entropy loss

Code: `cs336_basics/nn_utils.py`

4.2 The SGD Optimizer

Analysis code: `cs336_basics/sgd_analysis.ipynb`

At lr=1, there is barely any change in the loss, but as we increase the learning rate, we see the loss decreasing somewhat at lr=10 and quickly going to zero at

lr=100. When the learning rate is too big (lr=1000), the loss goes to infinity (diverges).

4.3 AdamW

Analysis code: `cs336_basics/adamw_analysis.ipynb`

(a) **How much peak memory does running AdamW require?**

- Parameters:

$$
\begin{aligned}
transormer =& embedding + num\_layers \times trans\_block + ln + lm\_head \\
& embedding = vocab\_size \times d_{\text{model}} \\
& trans\_block = 2 \times ln + attn + ffn \\
& \quad ln = d_{\text{model}} \\
& \quad attn = 4 \times d_{\text{model}}^2 \\
& \quad ffn = 3 \times d_{\text{model}} \times d_{\text{ff}} = 12 \times d_{\text{model}}^2 \\
& trans\_block \approx 16 \times d_{\text{model}}^2 \\
& ln = d_{\text{model}} \\
& lm\_head = d_{\text{model}} \times vocab\_size \\
transformer \approx& 2 \times vocab\_size \times d_{\text{model}} + 16 \times num\_layers \times d_{\text{model}}^2
\end{aligned}
$$

- Activations:
  Let us count the input sizes for the listed blocks. For convenience, set $D = context\_length \times d_{\text{model}}$

$$
\begin{aligned}
total =& batch\_size \times (num\_layers \times trans\_block + ln + lm\_head + cross\_entropy) \\
& trans\_block = 2 \times ln + attn + ffn \\
& \quad ln = D \\
& \quad attn = qkv\_proj + qk\_mult + softmax + weighted\_v + out\_proj \\
& \quad\quad qkv\_proj = D \\
& \quad\quad qk\_mult = 2D \\
& \quad\quad softmax = num\_heads \times context\_length^2 \\
& \quad\quad weighted\_v = num\_heads \times context\_length^2 + D \\
& \quad\quad out\_proj = D \\
& \quad attn = 5D + 2 \times num\_heads \times context\_length^2 \\
& \quad ffn = w1\_w3 + silu + w2 \\
& \quad\quad w1\_w3 = D \\
& \quad\quad silu = context\_length \times d_{\text{ff}} = 4D \\
& \quad\quad w2 = context\_length \times d_{\text{ff}} = 4D \\
& \quad ffn = 9D \\
& trans\_block = 16D + 2 \times num\_heads \times context\_length^2 \\
& ln = D \\
& lm\_head = D \\
& cross\_entropy = context\_length \times vocab\_size \\
total =& batch\_size \times (num\_layers \times \left(16D + 2 \times num\_heads \times context\_length^2\right) \\
& + 2D + context\_length \times vocab\_size)
\end{aligned}
$$

- Gradients - 1 per parameter
- Optimizer state - 2 per parameter (the moments)

Total:

$$
\begin{aligned}
memory &= 4 \times (parameters + activations + gradients + state) \\
&= 4 \times (4 \times parameters + activations) \\
&= 4 \times activations + 16 \times parameters \\
&= batch\_size \times [4 \times (num\_layers \times (16 \times context\_length \times d_{\text{model}} \\
&\quad + 2 \times num\_heads \times context\_length^2) + 2 \times context\_length \times d_{\text{model}} \\
&\quad + context\_length \times vocab\_size)] \\
&\quad + [16 \times (2 \times vocab\_size \times d_{\text{model}} + 16 \times num\_layers \times d_{\text{model}}^2)]
\end{aligned}
$$

(b) **Instantiate your answer for a GPT-2 XL-shaped model to get an expression that only depends on the batch_size. What is**

the maximum batch size you can use and still fit within 80GB memory?

We would need approximately $15,318,454,272 \times batch\_size + 34,030,438,400$ bytes. With 80GB of GPU memory ($80 * 1024^3$ bytes), the maximum batch size would be 3.

(c) **How many FLOPs does running one step of AdamW take?**

AdamW's step() method does not perform any matrix multiplications; the number of FLOPs is proportional to the number of parameters (the constant is hard to estimate because of the sqrt operation, but it's not big). The total cost is dominated by the forward and backward passes: $FLOPs \approx 6 \times batch\_size \times context\_size \times param\_count$.

(d) **Assuming you are able to get 50% MFU, how long would it take to train a GPT-2 XL for 400K steps and a batch size of 1024 on a single A100?**

Using the estimate from part 3, we need approximately $4,430,995,456,000 \times 3 \times 1024 \times 400\text{K}$ FLOPs. To get the number of days, we need to divide it by $24 \times 60 \times 60 \times (19.5 \text{ TFLOPs} / 2)$. Thus, on a single A100, we would need roughly 6464 days, or 17.7 years!

4.4 Learning rate scheduling

Code: `cs336_basics/optimizer.py`

4.5 Gradient clipping

Code: `cs336_basics/optimizer.py`

5 Training loop

5.1 Data Loader

Code: `cs336_basics/data.py`

5.2 Checkpointing

Code: `cs336_basics/serialization.py`

5.3 Training loop

Code: `cs336_basics/train.py`

6 Generating text

Code: `cs336_basics/generate.py`

7 Experiments

7.1 How to Run Experiments and Deliverables

For experiment tracking, I have used:

- Hydra for configurations and hyperparameter sweeps.
- Weights and Biases

I started by getting code to run on a Macbook Air (M4) - run with typical uncalibrated parameter values, achieving the validation error of 1.9. Then I switched to H-100, using lambda.ai to rent a GPU. The initial run achieved the validation error of 1.465. The time for all runs on H100 was 25-30 minutes with compiled models.

When compiling with mixed precision (bfloat16), I got numeric issues (NaNs) in the attention block. Using "aot_eager" compilation with bfloat16 resulted in much slower training and higher losses (https://wandb.ai/dmitry-frumkin-personal/cs336-2025-a1/runs/57g77up1). I carefully reviewed and debugged my code, but in the end had to skip mixed precision while still using "inductor" compilation on CUDA.

7.2 TinyStories

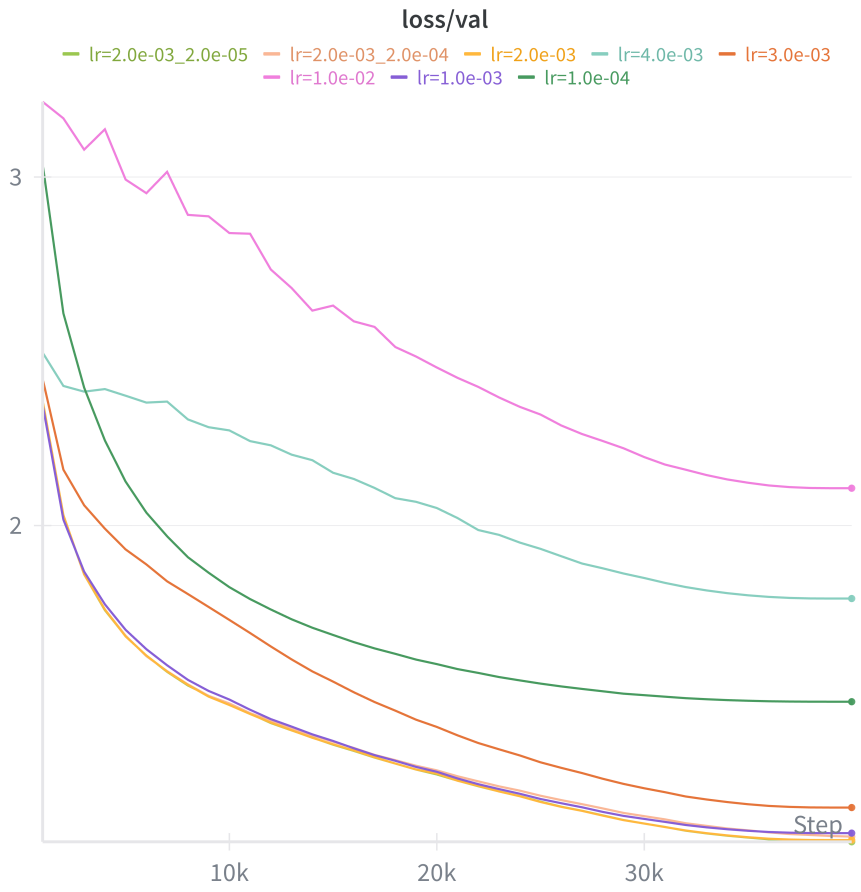
I used the following typical hyperparameter values:

- Gradient clipping: Max L2 norm = 1.0
- AdamW: betas = (0.9, 0.95), weight decay = 1e-2
- Cosine scheduling: warmup = max(100, round(0.02 * num_steps)) - mostly 800
- Batch size = 32

Tune the learning rate

(a) **Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).**

The initial test was with rather arbitrary max learning rate of 3e-4 and min learning rate 0.01 of that, 3e-6. For testing different learning rates, I still used cosine annealing, but set the minimum learning rate to 0. I logged validation loss every 1000 iterations. I started with 1e-2, 1e-3, 1e-4. 1e-3 was clearly the best of the three. The sweet spot was at 2e-3. Then I tried slightly higher minimum learning rates of 2e-4 and 2e-5 and got the best validation loss of 1.384 with the maximum learning rate of 2e-3 and the minimum 2e-5.


loss/val
lr=2.0e-03_2.0e-05
lr=2.0e-03_2.0e-04
lr=2.0e-03
lr=4.0e-03
lr=3.0e-03
lr=1.0e-02
lr=1.0e-03
lr=1.0e-04
3
2
Step
10k
20k
30k

(b) **Folk wisdom is that the best learning rate is "at the edge of stability." Investigate how the point at which learning rates diverge is related to your best learning rate.**

I still used cosine annealing, so there was no real divergence for learning rate values up to 1e-2. At the same time, there was clearly instability in the beginning and the shapes of the curves for learning rates 1e-2 and 4e-3 clearly indicate some instability in the beginning before cosine annealing kicks in. Given the big difference between the curves for 3e-3 and the optimal 2e-3, the folk wisdom seems generally correct; however, we have to bear in mind the effect of the scheduler.

Batch size variations

The above runs were with batch size 32. For other batch sizes, I have used square-root scaling of the learning rate:

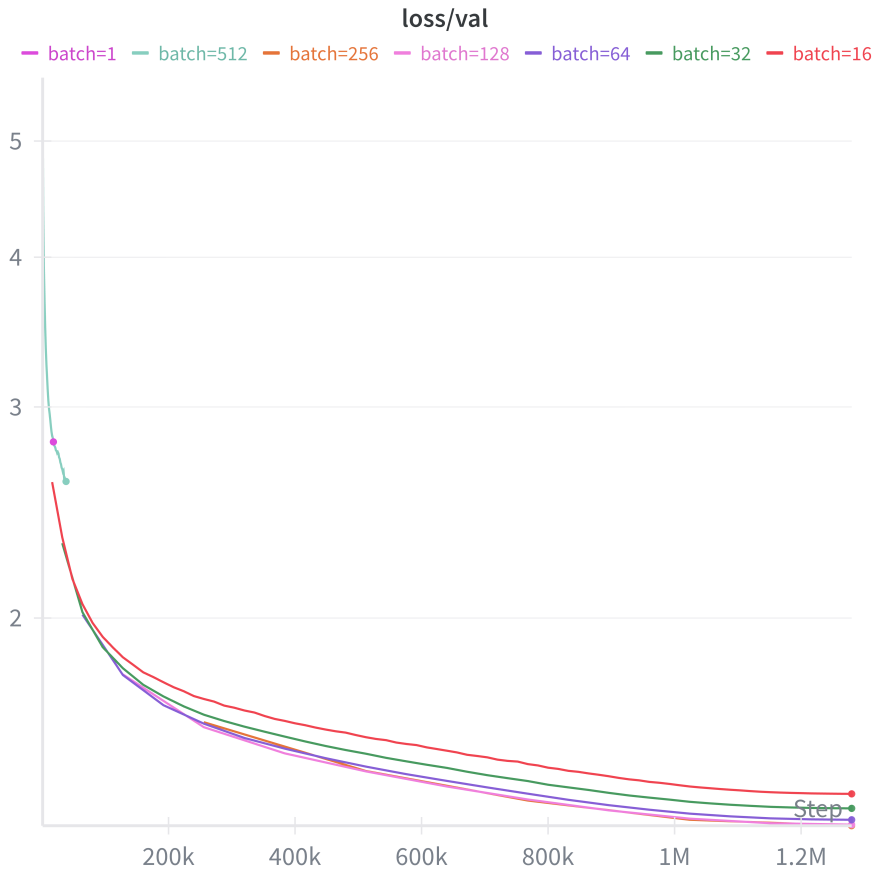
$$
\begin{aligned}
&base\_batch\_size = 32 \\
&base\_max\_lr = 2e-3 \\
&max\_lr = base\_max\_lr * \sqrt{\frac{batch\_size}{base\_batch\_size}} \\
&min\_lr = 0.01 \cdot max\_lr
\end{aligned}
$$

I have tried batch sizes 1, 16, 32, 64, 128, 256, 512. Training for batch sizes 1 and 512 was projected to take many hours and the training cuves looked bad, so I aborted those runs. Note: I changed the step axis to represent the sequence count to compare the curves.

In terms of speed, it took 17m4s for 64, 15m28s for 128, 31m7s for 256. Longer and shorter batches required more time: for small batches GPU is not utilized efficiently, but for large batches there could be a problem with GPU cache usage. Perhaps kernels are also optimized for certain block sizes.

The best validation loss of 1.342 was achieved for batch size 256 with 128 a very close second (1.345). Note that the number of tokens processed was very slightly different (must be a multiple of the batch size). The difference in the validation loss could be caused by different step sizes, different scheduler behavior (warm-up defined as 2% of the total number of steps), and also the imprecise nature of the square-root scaling of the learning rate.

The sweet spot is around batch size 128 with the maximum learning rate of 4e-3, but we could probably fine-tune both of these parameters (or at least the learning rate) to reduce the loss even further.



Generate text

Given the prompt "I think I did it again" with temperature = 0.7 and top_p = 0.9, I got:

> I think I did it again and said, "It's okay. I forgive you. But next time, please ask me first before you touch something new." The old man nodded and said, "Okay. I'll be careful. Thank you for letting me see your ball." The little girl smiled and said, "You're welcome, sir. I'm glad you are okay. And I'm glad you are safe." The old man smiled and said, "You're welcome. You're good kids. Now, let's go home and have some lunch."

The model tends to regurgitate phrases and sentences from the training set; thus, given a prompt that is out of distribution, like the one above, it makes a not always smooth transition to a familiar domain and continues from there. For example, another run with the same parameters resulted in:

> I think I did it again smiled. They knew that their love was special. From that day on, Mia always remembered to take care of her toys. And they all lived happily ever after.

When I decreased the temperature to 0.3, the model started having problems returning decodable sequences. Eventually, it produced this result:

> I think I did it again was silly. I did not listen to you. I did not listen to you. I did not respect you. I did not respect you. I was not a good friend. I was a bad friend. I was not a good friend. I was silly. I was silly. I did not listen to you. I did not care. I was silly. I was not a good friend. I was silly. I was not a good friend. I was silly. I was silly. I was silly. I did not listen to you. I did not care. I did not listen. I did not respect you. I did not care. I was a bad friend. I was a bad friend. I was not a good friend. I was silly. I was silly. I was not a good friend. I was silly. I was silly. I was not a good friend. I was silly. I was not a good friend. I was silly. I was not a good friend. I was silly. I was not a good friend. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I was silly. I was not a good friend. I was silly. I was silly. I was silly. I was not a good friend. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I did not care. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I did not care. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I did not care. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I was silly. I was silly. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I was silly. I was silly. I was silly. I was silly. I did not listen. I did not care. I was silly. I was silly. I was

Clearly, creativity has decreased, but the connection between the out-of-distribution prompt and the rest of the text is still bad. If I change the prompt to the familiar "Once", the model has no problem producing good coherent texts; for example:

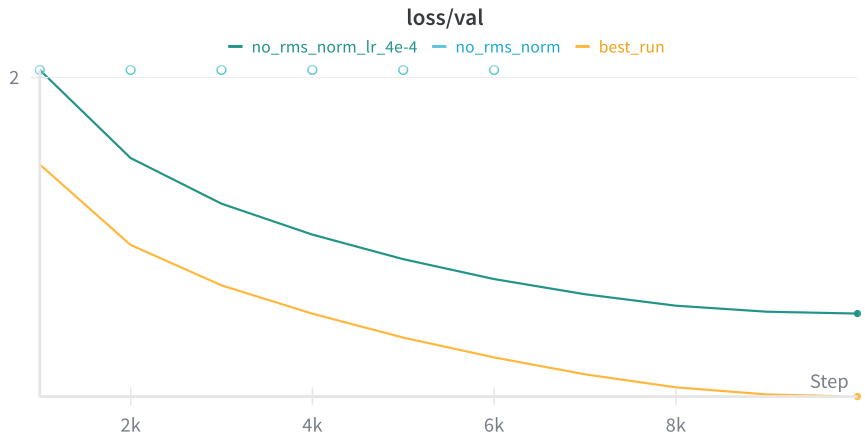
> Once, in a small town, there was a little girl named Lily. She had a toy cat that she loved very much. The cat was soft and had a big smile on its face. Lily and her cat played all day long. One day, Lily was playing with her cat when she saw a big dog. The dog looked at Lily and said, "Why are you playing with a cat?" Lily was scared, but she wanted to be brave. So, she said, "I will be your friend, cat. We can play together." Lily and the cat played all day. They ran, they jumped, and they laughed. They were very happy to have each other as friends. At the end of the day, Lily said, "Thank you, cat, for playing with me and my cat." The cat replied, "You're welcome, Lily. I had fun too. Let's play again soon." And they did.

The fluency of the output is affected by the prompt being familiar to the model and the temperature not being too high; for example, when I set it to 5, I got a complete word salad. On the other hand, if it's small, the output becomes boring. Top-p also acts as a safety net to prevent occasional bad results.

7.3 Ablations and architecture modification

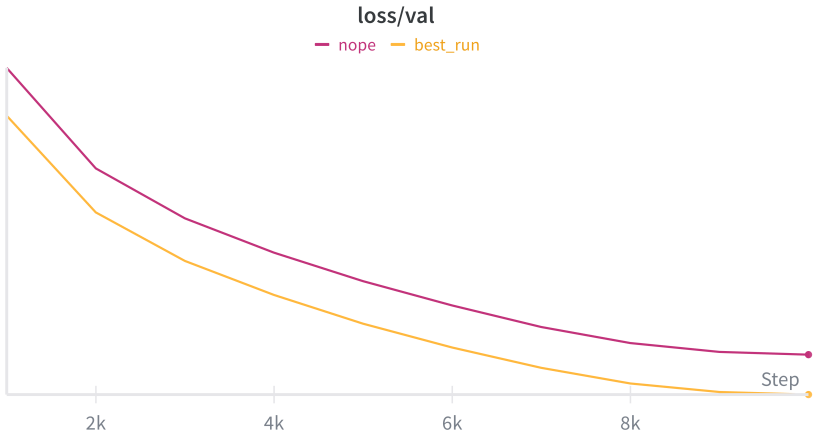
Remove RMSNorm and train

With the best parameters found so far (batch size 128, learning rate 4e-3), removing RMSNorms resulted in training diverging right away. With a reduced learning rate of 4e-4, training stabilized, but the validation loss was higher (1.49 in the end).



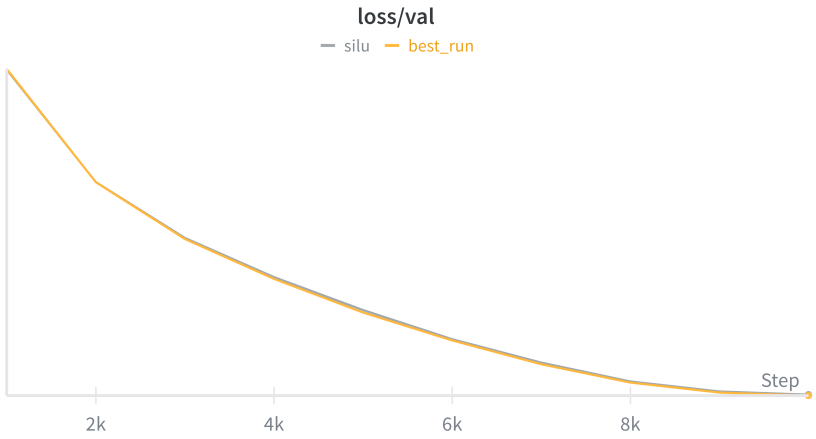
Implement post-norm and train

Implement NoPE



Without positional encoding the loss expectedly went up, but not too high (1.400 vs. 1.343). The model still manages to learn some positional information because of masking. It also makes use of statistical structure of the data (even without order) to make intelligent guesses about the next token.

SwiGLU vs. SiLU



Replacing SwiGLU by SiLU made virtually no difference (got validation loss 1.344 instead of earlier 1.343). The reason is probably that SwiGLU is supposed to outperform SiLU for larger models trained on larger datasets and in our toy example it does not matter what to use as long as the number of parameters is about the same.

7.4 Running on OpenWebText

I got a much higher final validation loss: 3.96 vs. 1.343 on TinyStories. With a much larger vocabulary (32,000 vs 10,000 tokens), the task is significantly harder, so the larger loss is not surprising.

Here is sample text generated from the prompt "I think I did it again":

> I think I did it again've been playing some of the best games of my career, but it's not about as much as I can. I'm still playing a lot of different games.
>
> I've been playing several different games in different parts of the world, and I've played that games in a different way. I've been playing a lot for a long time and I've been playing a lot.
>
> The first game I played was a little bit of a bit of a game. I played it for a while, but I've been playing it for years now.
>
> It's been a long time since I started playing games, and I think I'm ready to play it. I'm not very confident in playing this game.
>
> I think I have a lot of experience with my career. I think I've been playing the same game and I think it's a really interesting experience.
>
> "I've always been in the games where I can play and I've always been a part of it.
>
> "I've always been in the most difficult part of it. I've always been the most productive part of my life, and I'm still a great player. I always want to be a part of it.
>
> "I was so much focused on that. I was like, 'I'm a very good player.'
>
> "I've always been a big part of it. I think I'm a little bit older and I'm a little older and I feel like I've had a lot of time and I feel like I can play a lot of different games and then I'm going to have a lot of different things to say.
>
> "I'm very good at it. I like to play and I love to play. I think I like it."
>
> He's playing the same game as the others. He's got a lot of different things with it. He's always been a great player and a great player. He's a really good player and he's always kind of the same, but I think he's one of the most valuable players in the game. He's been a great player and he's not the same as me. He's a very good player and he's been great, I'm a great player, I can be a great player and I have great players who can make a great player.
>
> "He's a great player and I like him. He's a good

The text looks like English, but is not very coherent. The transition from the prompt is bad. In addition to the original 4e-3, I tried the learning rates of 1e-2 and 1e-3 without much success. Perhaps, we could get a minor improvement with more learning rate and batch size fine-tuning; however, the complexity of the text and the much larger vocabulary would likely require a much longer training with a longer context and a larger model to get decent results.

7.5 Your own modification + leaderboard

Given resource limitations (I am not a Stanford student), I skipped the OpenWebText leaderboard part.

On TinyStories, I tried weight tying, namely forcing the embedding layer and the linear head to share weights. During weight initialization I used the middle value of $\sigma = \sqrt{\frac{1}{d_{\text{model}}}}$. As a result, the final validation loss decreased from 1.343 to 1.333 (run).

In addition, I removed weight decay for the embedding layer (there was no reason for weight decay on embedding even without weight tying), which resulted in the final validation loss of 1.306 (run).