stanford336

echen333

July 31, 2025

1 Byte Pair Encoding

Problem : unicode1

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

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

(c) What happens when this character occurs in text? It may be helpful to play around with the following in your Python interpreter and see if it matches your expectations:

```
>>> chr(0)
>>> print(chr(0))
>>> "this is a test" + chr(0) + "string"
>>> print("this is a test" + chr(0) + "string")
```

Solution

(a) It returns '\x00', the null character.

(b) Printed representation is blank line, repr() is the actual hex.

(c) It prints the string representation where null character is not visible.

Problem : unicode2

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32? It may be helpful to compare the output of these encodings for various input strings

(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'
```

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

Solution

(a) UTF-8 makes vocab size smaller to 256 instead of 65k.

(b) It is unable to decode the japanese font こんにちは because it assumes each byte is a unicode character able to decoded, which is not true for most unicode characters.

(c) It cannot decode b'\xe3\x81' as it truncates b'\xe3\x81\x93' (こ).

Problem : train_bpe_tinystories

(a) Train a byte-level BPE tokenizer on the TinyStories dataset, using a maximum vocabulary size of 10,000. Make sure to add the TinyStories <|endoftext|> special token to the vocabulary. Serialize the resulting vocabulary and merges to disk for further inspection. How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?
Resource requirements: 30 minutes (no GPUs), 30GB RAM
Hint You should be able to get under 2 minutes for BPE training using multiprocessing during pretokenization and the following two facts:

(1) The <|endoftext|> token delimits documents in the data files.

(2) The <|endoftext|> token is handled as a special case before the BPE merges are applied.

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

Solution

(a) Took 3 minutes. The longest tokens in the vocab are

```
   [(9379, b' responsibility'),
(9143, b' disappointment'),
(7160, b' accomplishment'),
```

which makes sense as these are probably words that come up a lot in stories and are long.

(b) Most of the time is spent searching for the tuples that contain the bytes to merge. We optimized the merging step so that "the only pair counts that change after each merge are those that overlap with the merged pair" and parallelized the rest by splitting our dictionary into chunks. It is harder to profile after parallelizing though.

Problem : train_bpe_expts_owt

(a) Train a byte-level BPE tokenizer on the OpenWebText dataset, using a maximum vocabulary size of 32,000. Serialize the resulting vocabulary and merges to disk for further inspection. What is the longest token in the vocabulary? Does it make sense?
Resource requirements: 12 hours (no GPUs), 100GB RAM

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

Solution

(a) Tokenizing on OpenWebText took approximately nine hours on GCP's c4-standard with 24vCPUs and 90GB of RAM. Only needed like 10GB of RAM though, and I think this could be done faster on a machine more optimized for disk I/O.

The longest tokens in the vocab are of the form '————————————————————————————————' and `\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3...'` which makes sense given there are lots of long dashes as separators in the internet.

(b) The OWT tokenizer has a great vocab space so most of its tokens are complete words, whereas Tinystories has a lot more short phrases or prefixes/suffixes, e.g. "airst" or "odox". OWT also just has a wider variety of tokens like NBA or Shutterstock or Deutsche.

Problem : tokenizer_experiments

(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)?

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

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

(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?

Solution

(a) The Tinystories tokenizer has a compression ratio of 4.06 and the OWT tokenizer has a compression ratio of 4.37.

(b) The compression ratio is smaller when using the TinyStories tokenizer from 4.418 -> 3.32.

(c) It took my tokenizer around 2 hours to go through 11GB of data, implying a throughput of $\approx 10^6 \frac{\text{bytes}}{\text{s}}$. That would imply it would 6 days to go through 825GB of text.

(d) Here, uint16 is a reasonable choice since we have uint max value (65k) is greater than our vocabsize (32k).

2 Transformer LM Architechture

Problem : transformer_accounting

(a) Consider GPT-2 XL, which has the following configuration:
vocab_size : 50,257
context_length : 1,024
num_layers : 48
d_model : 1,600
num_heads : 25
d_ff : 6,400
Suppose we constructed our model using this configuration. How many trainable parameters would our model have? Assuming each parameter is represented using single-precision floating point, how much memory is required to just load this model?

(b) Identify the matrix multiplies required to complete a forward pass of our GPT-2 XL-shaped model. How many FLOPs do these matrix multiplies require in total? Assume that our input sequence has context_length tokens.

(c) Based on your analysis above, which parts of the model require the most FLOPs?

(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?

(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?

Solution

(a) Let $a = \mathrm{d}_{\text{model}}, \mathrm{b} = \mathrm{d}_{\text{ff}}$. Each transformer block requires $2a + 4a^2 + 3ab = 4 \cdot 10^7$ parameters, so num_layers $\times\, 4 \cdot 10^7 = 2 \cdot 10^9$. The token embeddings and the final head each have vocab_size $\times$ a $= 8 \cdot 10^7$ parameters. Thus, the model would have $\approx 2.1 \cdot 10^9$ parameters, or 4GB worth of float16s.

(b) We examine the number of FLOPs per TransformerBlock as the embedding and final layer should be negligible. The multi-head self-attention consists of 4 projections of Q,K,V and back and self-attention. The projections, in total, cost

$$4 \cdot 2 \cdot (\text{seq}) \cdot (\text{d\_model})^2 \approx 2.1 \cdot 10^{10} \text{ FLOPs}$$

and the self attention part costs

$$\begin{aligned} &2 \cdot (\text{num\_heads}) \cdot (\text{seq})^2 \cdot (\text{d\_k}) + 2 \cdot (\text{num\_heads}) \cdot (\text{seq})^2 \cdot (\text{d\_k}) \\ &= 4 \cdot (\text{d\_model}) \cdot (\text{seq})^2 \approx 6.7 \cdot 10^9 \text{ FLOPs.} \end{aligned}$$

The FFN/ Swiglu requires 3 matrix multiplies per layer, costing

$$3 \cdot 2 \cdot (\text{seq}) \cdot (\text{d\_ff}) \cdot (\text{d\_model}) \approx 6.3 \cdot 10^{10}.$$

Computing the total number of FLOPs requires just multiplying by num_layers = 48 $\implies$ $1.3 \cdot 10^{12}$ FLOPs for MHSA and $3 \cdot 10^{12}$ FLOPs for the FFN.

(c) The analysis above shows that the FFN requires the majority (around 75%) of the FLOPs.

(d) We list our computations in Table 1. As the model size increases, the FFN takes up proportionally less FLOPs given we hold d_ff constant.

(e) Then, each forward pass will need 10^{14} FLOPs for the MHSA and $5 \cdot 10^{13}$ FLOPs for the FFN. Now, the MHSA takes twice as many FLOPs as the FFN instead of a quarter because the MHSA scales quadratically with sequence length.

Model	**MHSA**	**FFN**
GPT-2 Small	$9.6 \cdot 10^{10}$	$3.6 \cdot 10^{11}$
GPT-2 Medium	$3.1 \cdot 10^{11}$	$9.6 \cdot 10^{11}$
GPT-2 Large	$6.8 \cdot 10^{11}$	$1.8 \cdot 10^{12}$

Table 1: Transformer architecture components by GPT-2 model size

3 Training a Transformer LM

Problem : adamwAccounting

Let us compute how much memory and compute running AdamW requires. Assume we are using float32 for every tensor.

(a) How much peak memory does running AdamW require? Decompose your answer based on the memory usage of the parameters, activations, gradients, and optimizer state. Express your answer in terms of the batch_size and the model hyperparameters (vocab_size, context_length, num_layers, d_model, num_heads). Assume d_ff = 4 × d_model. For simplicity, when calculating memory usage of activations, consider only the following components:

- Transformer block:
  - RMSNorm(s)
  - Multi-head self-attention sublayer: QKV projections, Q K matrix multiply, softmax, weighted sum of values, output projection.
  - Position-wise feed-forward: W1 matrix multiply, SiLU, W2 matrix multiply
- final RMSNorm
- output embedding
- cross-entropy on logits

Deliverable: An algebraic expression for each of parameters, activations, gradients, and optimizer state, as well as the total.

(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? Deliverable: An expression that looks like a · batch_size + b for numerical values a, b, and a number representing the maximum batch size.

(c) How many FLOPs does running one step of AdamW take? **Deliverable**: An algebraic expression, with a brief justification.

(d) Model FLOPs utilization (MFU) is defined as the ratio of observed throughput (tokens per second) relative to the hardware's theoretical peak FLOP throughput [Chowdhery et al., 2022]. An NVIDIA A100 GPU has a theoretical peak of 19.5 teraFLOP/s for float32 operations. 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? Following Kaplan et al. [2020] and Hoffmann et al. [2022], assume that the backward pass has twice the FLOPs of the forward pass. Deliverable: The number of days training would take, with a brief justification

Solution

(a) From 2(a), we determined that each transformer block requires $2a+4a^2+3ab$ parameters, where a = d_model, b = d_ff. Also, the final RMSNorm contributes another a and the

output embedding contributes $a \cdot (\text{vocab\_size})$ parameters.

The activations in each transformer block require $7 * \text{batch} * \text{seq} * \text{d} + 2 * \text{batch} * (\text{seq})^2$ floats for the MHSA, $3 * \text{batch} * \text{seq} * (4\text{d}) + \text{batch} * \text{seq} + \text{d}$ float for the FFN, and another $4 * \text{batch} * \text{seq} * \text{d}$ floats for the residuals and norm activations. (Something like this in terms of OOM, I'm not being very careful.) The final output embedding activation also requires $\text{batch} * \text{seq} * \text{vocab\_size}$ floats. Also, the activations for the final RMSNorm and cross-entropy on logits should be negligible.

For the gradient and optimizer state, note that for each parameter, we also need 1 parameter for the gradient and 2 parameters for the AdamW m and v states.

Therefore, for the total number of floats we need, we get:

- Parameters: $\text{num\_layers} * (2\text{a} + 4\text{a}^2 + 3\text{ab}) + (\text{a} * \text{vocab\_size}) + \text{a}$
- Activations: $23 * \text{batch} * \text{seq} * \text{d} + 2 * \text{batch} * (\text{seq})^2$
- Gradient and optimizer states: 3 * # params.

(b) Plugging in the values for GPT-2 XL, we get:

- Parameters: $2.1 \cdot 10^9$ as shown in 2(a).
- Activations: $4.0 \cdot 10^7 * (\text{batch})$
- Gradient and optimizer states: $6.3 \cdot 10^9$.

Thus, we get the following expression for the peak amount of memory used: $4 * (8.4 \cdot 10^9 + 4.0 * 10^7 * \text{batch\_size})$. Then, the maximum batch size on 80GB would be 290.

(c) Running one step of AdamW takes ≈ 20 FLOPs per parameter since we need to compute the rolling first and second moments and perform weight decay. I am assuming each sqrt operation takes one FLOP though.

(d) To train GPT-2 XL, we need $\approx 4 \cdot 10^{12}$ FLOPs for each forward pass, as shown in 2(a). Then, we need $4 \cdot 10^{12} * 400k * 1024 \approx 1.6 \cdot 10^{21}$ FLOPs total for all 400k steps of the forward pass, implying $4.8 \cdot 10^{21}$ FLOPs total. Then, the total number of days training would take would be

$$\frac{4.8 * 10^{21}}{19.5/2 * 10^{12}} * \frac{1}{3600} * \frac{1}{24} = 5700 \text{ days.}$$

4 Experiments

Problem : learning_rate

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).
Deliverable: Learning curves associated with multiple learning rates. Explain your hyperparameter search strategy.
Deliverable: A model with validation loss (per-token) on TinyStories of at most 1.45

(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.
Deliverable: Learning curves of increasing learning rate which include at least one divergent run and an analysis of how this relates to convergence rates

Solution

(a) We just focused on tuning cosine lr and compared loss curves across many experiments. It helped to sweep over one parameter at a time. We basically just fixed T_final =40,000 and $\alpha_{\text{min}} = 1e-5$ and only tuned α_{max} and T_{warmup}. On run 89, we achieve a validation loss of 1.43.

(b) This seems true – too small of a learning rate would converge too quickly at a high loss. The larger learning rates that do not diverge worked well and are pretty close to our optimal learning rate of $3 \cdot 10^{-3}$.

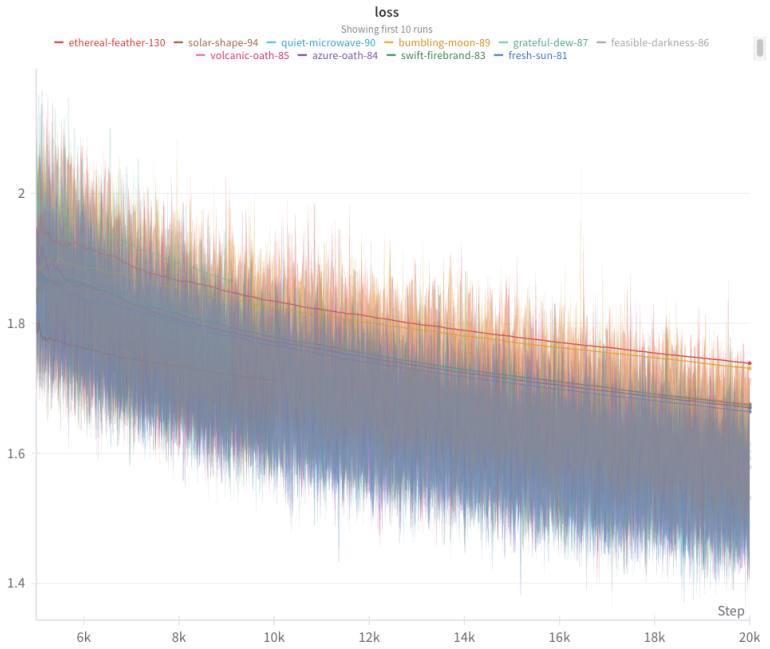


Figure 1: Learning rate vs loss.

Problem : batch_size_experiments

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.
Deliverable: Learning curves for runs with different batch sizes. The learning rates should be optimized again if necessary.
Deliverable: A few sentences discussing of your findings on batch sizes and their impacts on training.

Solution

Doubling batch size, time per batch becomes a little less than double batch sizes when doubling from 32 to 64 to 128 to 256. We also see a doubling of batch size does decrease loss faster, but not as quickly as 2x the steps. We run out of memory with a batch size of 1024, so batches of size >100 seem unfeasible with a larger model. But larger batch sizes like 128, 256, and 512 do seem promising compared to 64, though it is hard to say without tuning learning rates for both. Also, we note changing batch of size from 16 to 1 doesn't decrease speed too much, which is expected due to bottlenecks by I/O.

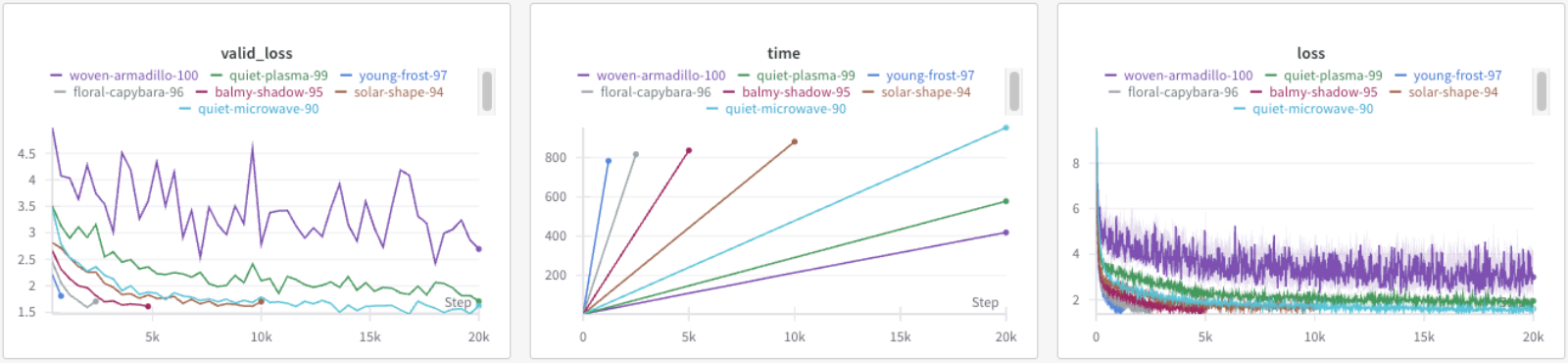


Figure 2: Batch size comparisons where num_steps is x-axis.
Batch sizes for runs 94-100 are [64,128,256,512,1024,16, 1]

Problem : generate

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (temperature, top-p, etc.) to get fluent outputs.
Deliverable: Text dump of at least 256 tokens of text (or until the first <|endoftext|> token), and a brief comment on the fluency of this output and at least two factors which affect how good or bad this output is.

Solution

Once upon a timi, there was a little girl named Lucy. She loved to play with her toys. One day, Lucy found a big, shiny rock. She thought it was very pretty. Lucy showed the rock to her mom. Her mom said, "That rock is so pretty! Let's put it in our garden." Lucy was happy and put the rock in her garden. Every day, Lucy and her mom would go to the garden.

They would look at the rock and smile. Lucy loved her new rock. She knew it was special. <|endoftext|>

Here, the output seems pretty fluent, given that we use load a model with low loss and use temperature = 0.

4.1 Ablations

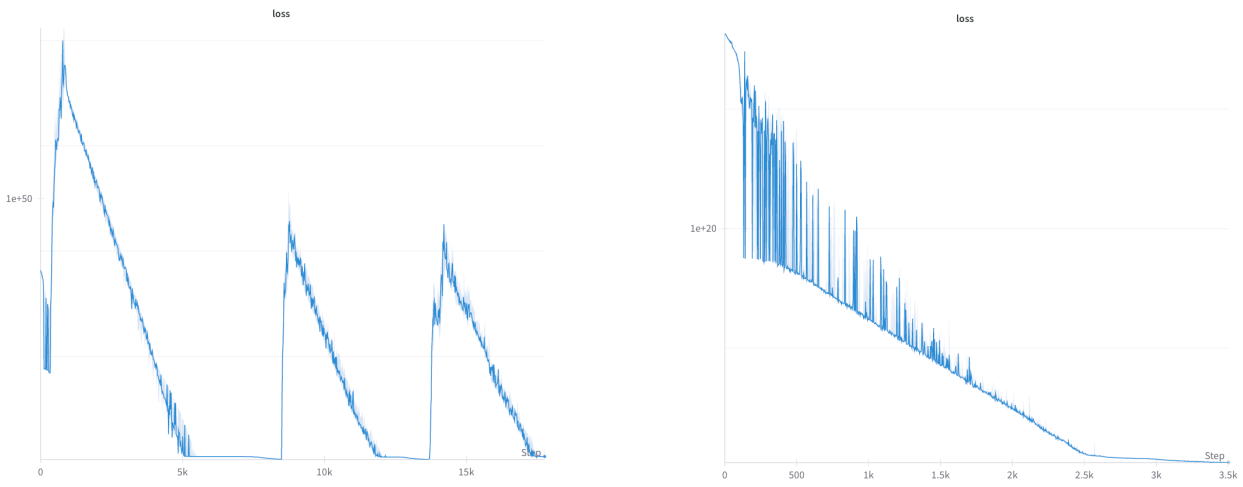
Problem : layer_norm_ablation

Remove all of the RMSNorms from your Transformer and train. What happens at the previous optimal learning rate? Can you get stability by using a lower learning rate?
Deliverable: A learning curve for when you remove RMSNorms and train, as well as a learning curve for the best learning rate.
Deliverable: A few sentence commentary on the impact of RMSNorm.

Solution

Without layer norm, our loss is very large. With our initial learning rate, we have that loss decreasing initially from 10^{35} to 10^{20}, then increases to 10^{74}, then decreases to 5, then diverges, and so on. With a lower learning rate of 10^{-3} and $\alpha_{\text{min}} = 0$, we are able to get convergence with early stoppage.

Note: both graphs have y-axis log scaled.



(a) Initial best learning rate

(b) Best learning rate

Figure 3: No layer norm loss curves with different learning rates.

Problem : pre_norm_ablation

Modify your pre-norm Transformer implementation into a post-norm one. Train with the post-norm model and see what happens.

Deliverable: A learning curve for a post-norm transformer, compared to the pre-norm one.

Solution

It is basically the same, but with a lot of smoothing, we see that the post-norm transformer has a tiny bit higher loss than pre-norm.

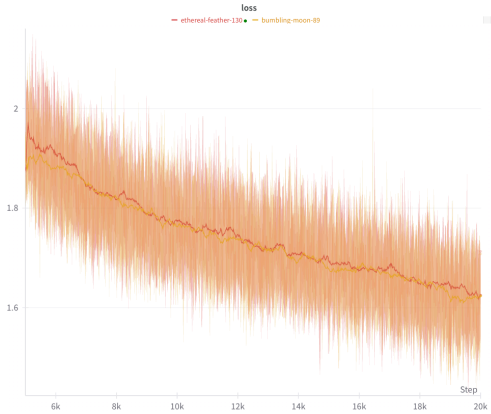


Figure 4: Comparison of pre and post-norm loss curves.
Red denotes post-norm, yellow pre-norm.

Problem : no_pos_emb

Modify your Transformer implementation with RoPE to remove the position embedding information entirely, and see what happens.
Deliverable: A learning curve comparing the performance of RoPE and NoPE.

Solution

Without RoPE, we see that the loss is a bit higher, as shown in Figure 5.

Problem : swiglu_ablation

Deliverable: A learning curve comparing the performance of SwiGLU and SiLU feed-forward networks, with approximately matched parameter counts.
Deliverable: A few sentences discussing your findings.

4.2 Running on OpenWebText

Problem : main_experiment

Train your language model on OpenWebText with the same model architecture and total training iterations as TinyStories. How well does this model do?
Deliverable: A learning curve of your language model on OpenWebText. Describe the

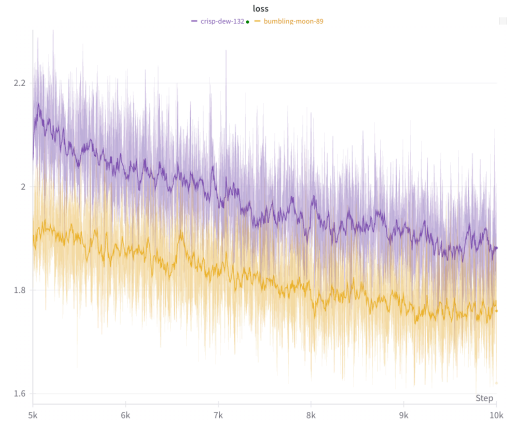


Figure 5: Comparison of RoPE and NoPE loss curves.
Purple denotes no RoPe, yellow has RoPE.

difference in losses from TinyStories – how should we interpret these losses?
Deliverable: Generated text from OpenWebText LM, in the same format as the TinyStories outputs. How is the fluency of this text? Why is the output quality worse even though we have the same model and compute budget as TinyStories?

Solution

It was probably not a good idea to use temperature 0, but we still get worse quality as OWT is large and has a lot of unfiltered / bad data that is being trained on I presume. In tinystorise, we had a small amount of high quality data. With the starting prompt "Once upon a ti", we get:

Once upon a tiara, the first time I saw a picture of a man in a white shirt, I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt. I saw a man in a white shirt, and I saw a man in a white shirt...

Problem : leaderboard

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within 1.5 H100-hour.
Deliverable: The final validation loss that was recorded, an associated learning curve that clearly shows a wallclock-time x-axis that is less than 1.5 hours and a description of what you did. We expect a leaderboard submission to beat at least the naive baseline of a 5.0 loss. Submit to the leaderboard here: https://github.com/stanford-cs336/assignment1-basics-leaderboard.

Solution

We achieve a final validation loss of 3.6 with a few changes from our original tinystories model – mostly hyperparameter tuning, switching to bfloat16, and using torch.compile. There are still many optimizations to be made though, which we hope to revisit soon.