CS 336: Language Modelling from Scratch

Assignment 1

April 16, 2024

SUNet ID: mattreed
Name: Matt Reed

1 unicode1

(a) What unicode character does char(0) return?

It returns the null character.

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

When printed, it usually doens't show up at all. However, repr will print out its byte representation: “\\ x00”.

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

Usually nothing. When the text is printed it is simply ignored. Python doesn't use null terminated strings so it doesn't affect much.

2 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.

UTF-16 and UTF-32 have byte prefixes for even simple bytes like the alphabet. This means we'd have to encode up to 4 times as many bytes if we used UTF-32 instead of UTF-8. We'd end up compressing these prefix bytes anyway, so we might as well make it more interpretable and efficient and use UTF-8.

(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.

The function is incorrect because some characters in unicode are multiple bytes long. This function decodes each byte individually which wouldn’t make sense because some bytes need to be decoded together to make a character. If I inputted an emoji for instance, it would not work at all and would interpret it as a few different bytes.

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

“\F0\x9F”, which is the prefix for the four byte sequence for the smiley face emoji: “\F0\x9F\x98\x81”

3 train_bpe_tiny_stories

(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: N hours (no GPUs), N GB RAM Deliverable: A one-to-two sentence response.

It took 18.9 minutes and 5 GB to train the tokenizer of tiny-stories-train.txt on my M3 chip. The longest token is ” accomplishment” which makes sense because its a common long word.

(b) Profile your code. What part of the tokenizer training process takes the most time? Deliverable: A one-to-two sentence response.

The pretokenizing and counting pretokens each take around 16% of the total time with the merging and updating pair counts taking the longest time (around 70%).

4 train_bpe_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? Deliverable: A one-to-two sentence response.

The longest token in my vocabulary is 8 em-dashes in a row (which is 24 bytes). This makes sense because I didn't have time to train a perfect OWT tokenizer so I made the maximum pretoken length 8.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText. Deliverable: A one-to-two sentence response.

The tiny stories tokenizer focussed much more on common words and encoding words whereas the OWT tokenizer had much more random data so it compressed weirder combinations of byte sequences that appea on the internet.

5 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)? Deliverable: A one-to-two sentence response.

The OWT tokenizer has a compression ratio of 4.22 on OWT and 3.90 on Tiny Stories. The Tiny Stories tokenizer has a compression ratio of 4.04 on Tiny Stories and 3.41 on OWT.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens. Deliverable: A one-to-two sentence response

As shown in the compression ratios, the tiny stories tokenizer does worse on the OWT text than the OWT tokenizer (3.41 vs. 4.22). This is because it has a smaller vocabulary that was only trained on simple words, so it isn't prepared for the vastness in byte sequences from OWT.

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

I was able to encode 2.2GB of text in 40 minutes which means it has a throughput of about 900,000 bytes/s (including both the pretokenizing and merging). This means it would take about 249 hours to encode the Pile dataset (or 10 days).

(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?

uint16 is an appropriate choice because the maximum value of a uint16 is $2^{16} - 1 = 65535$ which is big enough to accommodate the 10,000 and 32,000 vocabularies of both tokenizers.

6 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? Deliverable: A one-to-two sentence response. 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.

Embedding Layers = 1600 x 50257 + 1024 x 1600

Transformer Layers = 48 x (

Norm-1 = 1600

mha = 4 x 1600 x 1600

Norm-2 = 1600

ff = 2 x 1600 x 6400

Output Norm = 1600

Output Projection = 50257 x 1600

Total = 1.637 billion

Which would take up 6.548 GB of space using float-32s

Matrix Multiplies:

q,v,k projections, and value up projection per head, per layer
FLOPs = 48 x 4 x 2 x 1600 x 1600 x 1024 = 1.006 trillion (0.29)
scaled dot product attention
FLOPs = 48 x 2 x 2 x 1024 x 1600 x 1024 = .322 trillion (0.09)
feed foward layer per layer
FLOPs = 48 x 2 x 2 x 1600 x 6400 x 1024 = 2.013 trillion (0.57)
and output projection
FLOPs = 2 x 1600 x 50257 x 1028 = .1646 trillion (0.04)
Total flops = 3.506 trillion

required.

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

The Feedforward layers take the most, followed by the Multihead Attention projections (and the scaled dot product attention mechanism), then the output layer.

(c) 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?

Small:
attention FLOPs = 12 x 4 x 2 x 768 x 768 x 1024 = 5.798e10 (0.11)
dot product FLOPs = 12 x 2 x 2 x 1024 x 768 x 1024 = 1.546e11 (0.29)
feed forward FLOPs = 12 x 2 x 2 x 768 x 768 x 1024 = 2.415e11 (0.45)
output FLOPs = 2 x 768 x 50257 x 1028 = 0.7904e11 (0.14)
Total flops = 5.332e11

Medium:
attention FLOPs = 24 x 4 x 2 x 1024 x 1024 x 1024 = 2.06e11 (0.18)

dot product FLOPs = 24 x 2 x 2 x 1024 x 1024 x 1024 = 2.06e11 (0.18)
feed forward FLOPs = 24 x 2 x 2 x 1024 x 1024 x 1024 = 6.44e11 (0.55)
output FLOPs = 2 x 1024 x 50257 x 1028 = 1.05e11 (0.09)
Total flops = 1.16e12

Large:
attention FLOPs = 36 x 4 x 2 x 1280 x 1280 x 1024 = 4.831e11 (0.23)
dot product FLOPs = 36 x 2 x 2 x 1024 x 1280 x 1024 = 2.57e11 (0.12)
feed forward FLOPs = 36 x 2 x 2 x 1280 x 1280 x 1024 = 1.207e12 (0.58)
output FLOPs = 2 x 1280 x 50257 x 1028 = 1.317e11 (0.063)
Total flops = 2.08e12

As the size of the model increases (without increasing context length), the scaled dot product attention takes up a smaller proportion of the total flops while the attention projections and feed forward layers take up more since they scale with the square of the model dimensions.

(d) 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?

q,v,k projections, and value up projection per head, per layer
FLOPs = 48 x 4 x 2 x 1600 x 1600 x 16,384 = 16.10 trillion (0.12)
scaled dot product attention
FLOPs = 48 x 2 x 2 x 16,384 x 1600 x 16,384 = 82.46 trillion (0.62)
feed foward layer per layer
FLOPs = 48 x 2 x 2 x 1600 x 6400 x 16,384 = 32.21 trillion (0.24)
and output projection
FLOPs = 2 x 1600 x 50257 x 16,384 = 2.63 trillion (0.02)
Total flops = 133.42 trillion

Increasing the context length significantly increases the flops required for scaled dot product attention because that process scales with the context length squared whereas the rest increases with it linearly.

7 Adam_W_Accounting

(a) How much 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 model hyperparameters (num_layers, d_model, context_length). Assume d_ff = 4 × d_model. Deliverable: An algebraic expression for each of parameters, activations, gradients, and optimizer state, as well as the total.

Total Memory=Parameters Memory+Activations Memory+Gradients Memory+Optimizer State Memory

Parameters Memory = d_model x vocab_size + context_length x d_model
+ num_layers x (
d_model
+ 4 x d_model x d_model
+ d_model
+ 2 x d_model x 4 x d_model
)
d_model
+ vocab_size x d_model

Activations Memory = batch_size x num_layers x (
(context_length x context_length) + (4 x d_model)
)

Gradients Memory = batch_size x Parameters Memory

Optimizer State Memory = 2 x Parameters Memory

(b) Instantiate your answer for GPT-2 XL 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

Total Memory = ((batch_size x 48 x ((1024 x 1024) + (4 x 1600)) + ((2 + batch_size) x 1.637 billion)) x 4 byte/float
Total Memory = 1.309e10 + batch_size * 6.548e9

$$80GB > 1.309e10 + \text{batch\_size} * 6.548e9$$
$$\text{batch\_size} < 10$$

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

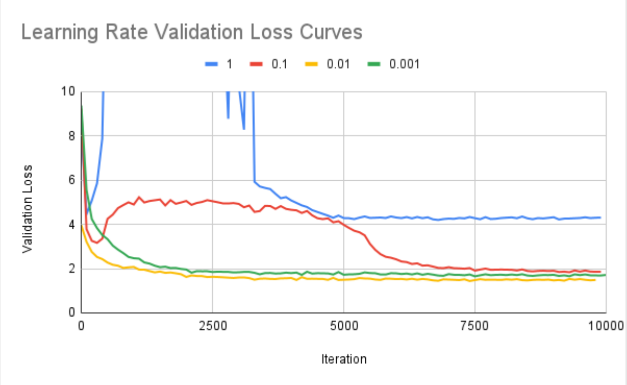
Twice the FLOPs of the forward pass, or 7.012 trillion.

(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

Time = (7.012e12 x 400K)/(0.5 x 19.5e12) = 287671 seconds, or about 3.3 days

8 Tuning learning rate

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



My approach was to keep multiplying my maximum learning rate by 10 until it started diverging. Because of the cosine annealing schedule, they all managed to return to a somewhat stable place in the end, however it is clear that the learning rate of 1 diverged severely and 0.1 was still too high. The learning rate of 0.01 preformed best, so that is what I will stick with. The final validation loss of that model was barely good enough at 1.45.

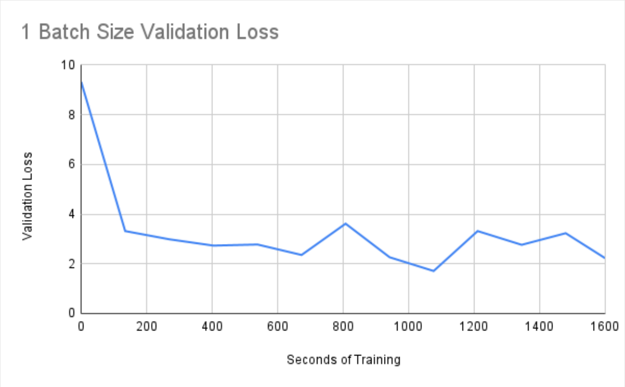
(b) Folk wisdom is that the best learning rate is “at the edge of stability.” Investigate how the point at which learning rates diverge is related to your best learning rate.

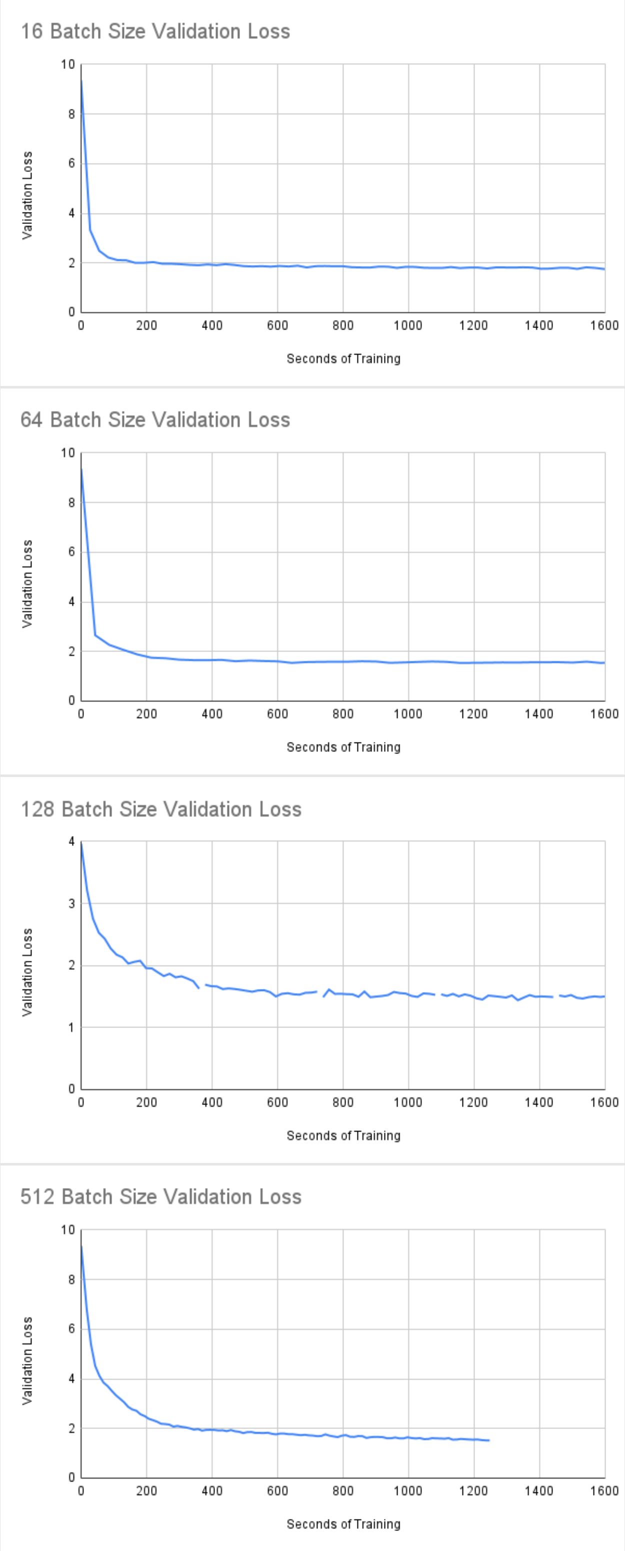
The learning rate that was closest to the first diverging rate (0.01 which was closest to 0.1) is the one that I will chose since it is on the brink of chaos. It empirically performed better than 0.001 even though that might seem counter intuitive. I believe the stochasticity of it allowed it to reach a better global optimum.

9 Batch Size Experiment

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

I ran 5 experiments each for 30 minutes. I trained with batch sizes of 1,16, 64, 128, and 512. The primary difference I noticed was training stability. With batch size of 1 (pure SGD), the validation loss was a lot more unstable, and that decreased with an increase in batch size. I know it’s diffiult to compare the graphs because I couldn’t figure out how to put them on the same figure when they have different time intervals and I don’t have any more patience to figure it out. Overall, the trend was clear, however, that the larger the batch size (512 is the largest that fits on the GPU), the lower the validation loss after 30 minutes. This makes sense because of the way that GPU’s are optimized, so I will stick with a large batch size of 256 or 512 for future training.





16 Batch Size Validation Loss

64 Batch Size Validation Loss

128 Batch Size Validation Loss

512 Batch Size Validation Loss

10 Generate Text

Use your decoder and your trained checkpoint, and see the text that comes out from 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.

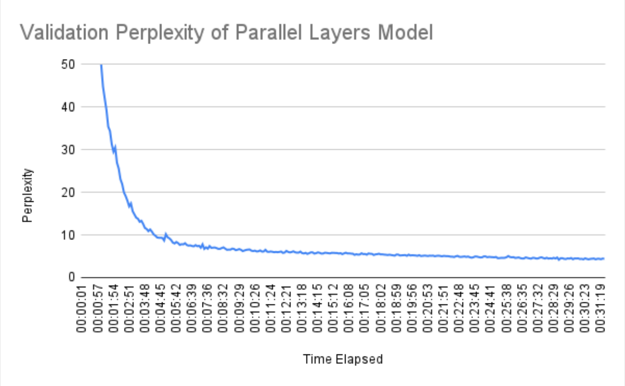
I ran three generations with different temperatures to see what I would get. The prompt I started the model with was “Once upon a time,”
Temperature 1.0: Once upon a time, there the cow garden loved very eager but and had It had had painting small leaves He was mailed scaring doneed big brother and popular from. They had no's play with a surprise again. It was inside. Follow inside that their pajamas showed, shiny asking was being loved to add saw that it. Then. He told Anna! Riri or. The bunny flowers was gone and couldn was more again. One day, Mia, ”Don some party and dry of the tree family their there was for a book a girl named Lily, counting friends out her mom and did animals unique on. She wanted its was just of it hello. One day, sad slowly. Tim. But the fence, in along this it went to believe on, while, Lila in the rock, Sam wanted to eat sorry with a big lion with warm and reached Tim's the heading Mittens on the... asked. He looked. It helps, ”Ready and his mum.

Temperature 0.5: Once upon a time, there there was a small there was a little boy there lived in the day, a little boy named Mia., there was very much to play with her be a big, there was a little girl was a small house and said, a big tree. They. One day long day, there was a big, ”I don very happy and said. She liked to play idea., ”I want to the table”

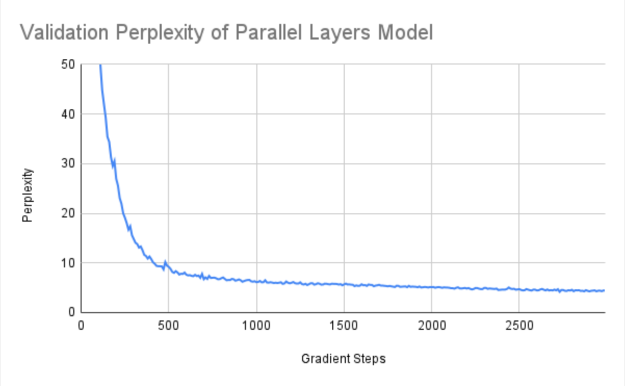
Temperature 0.1: Once upon a time, there there was a little boy named Tim was a big, there was a big tree. One day, there a big, there was a little girl named Tim was a big little girl named Tim was a little girl named Tim was a time, ”I'm sorry.

11 parallel_layers

Implement parallel layers, as defined in Eq 14. Train a model with parallel layers, logging your training curves. Deliverable: Two training curves associated with your parallel layer model. One where the x-axis is the number of gradient updates and y-axis is validation perplexity. Another is where the x-axis is wallclock time. Deliverable: Discuss any differences between the two training curves in a few sentences. Are the results surprising?



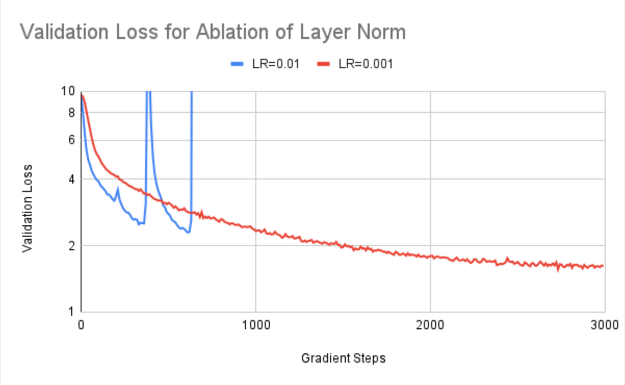
Validation Perplexity of Parallel Layers Model



Validation Perplexity of Parallel Layers Model

12 Layer Norm Ablation

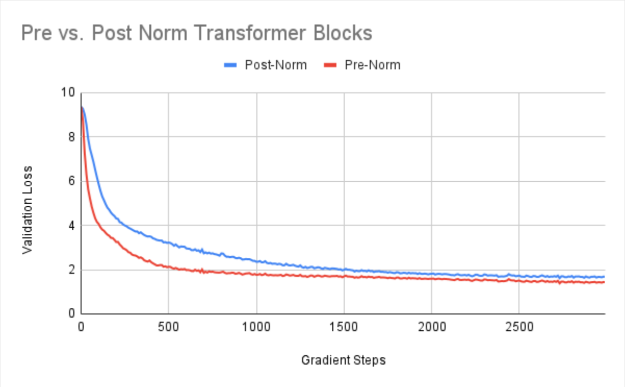
Remove all of the RMSNorms from your Transformer and train. What happens at the previous optimal learning rate? Can you get stability by using a lower learning rate? Deliverable: A learning curve for when you remove RMSNorms and train, as well as a learning curve for the best learning rate. Deliverable: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



Validation Loss for Ablation of Layer Norm

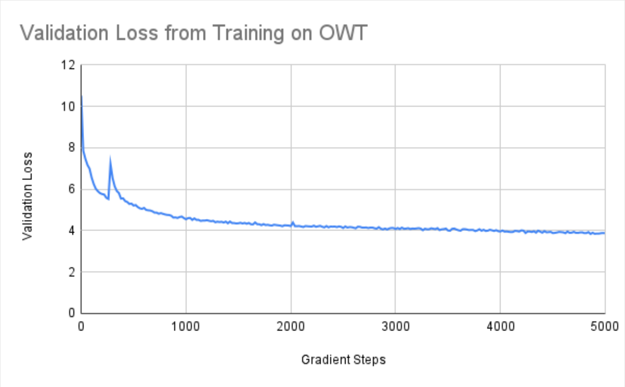
13 Pre Norm Ablation

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.



14 Main Experiment

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



Since the vocab size of the OWT model is larger, we can't necessarily compare the loss between this model and the tiny stories model. We could maybe exponentiate the loss and multiply it by the ratio in vocab sizes so they are roughly the same order of magnitude, but I don’t think this would even give us something to compare meaningfully.

Below I have sampled some text from the model with the prompt “the ” just to see where the model would take me. With a value of 0.7, I got some pretty decent results. However, it is still definitely less fluent than the tiny stories model, likely because it has so many non-word tokens in it’s dictionary.

Temperature 1.0: The 0000 14.3 KB ¿ 8.8.xbin bar 10 — old settings — gm/RunningBuffer.c/Windows 7 — support.c/Unfortunately xcli 2.1832.0 227.8.86616.114831731.33021.18861332.1879629 He added variable file? Very nice. I’ve been using nano-loops and rustc. Plus I’ve got an anonymous version seven. SIGERMMOROOLETE Full disclosure

Doping recording External Comments

Want to search and do that? Well, I call it a SIGED *dogFutures . . . . See “Fuelbox.”

I open the terminal with

- Jones Wt! Has been on great command include. In my interview, I have warned everyone who forgets you will have sufficient swiftness to dare read some of those who can help. What help silence one can skepticate can that please ensure your pacemaker would no longer be . . . Don’t read that. — Lloyd Jolardman (@OFFFFFFS), @SehenbergJolardman. WtRescu Julian Fellager (@TDaypridd) October 18, 2017

Simple Now guys and are not interested in platonic improvements in the future ”

Temperature 0.7: The vernment-loving man, the middle-class fighter of the world, a young man who has never had a life of believer in his life or, when, as something has been promised and had no real life.

”I have no concept in my life, I don’t know. I didn’t understand, I told my wife, ’That is the part of a human story... I would believe it, though. I would not be satisfied with it. I would not be be able to be able to see it.’”

Umbracing his father’s son, a former fighter, a former military-intelligence officer, was a free agent of his time in the field, at the same time that the Pentagon had repeatedly made a terrorist move to the piece.

The Pentagon loves these guys to have his own life.

”It was [the] dead of the war, but we didn’t have to take our own lives,” said Defense Secretary Leon Panetta.

”He was not a doctor. But if you look at this guy, he would not be able to use his life to hide his life as a public enemy.”

”I can’t believe it,” said Panetta, who now lives in New Jersey at the Pentagon’s residence. ”If he is dead, he’s not alive. He will never be able to live. He is a doctor.”

Th

Temperature 0.2: The vernote is a very simple, very simple, simple, and easy to use.

The first thing you want to do is use the vernote.com to create a new vernote.js file. This is a simple example of the vernote.js file.

The vernote.js file is a simple example of the vernote.js file.

The vernote.js file is a simple example of vernote.js file.

The vernote.js file is a simple, simple, simple, simple, and easy to use. It’s a great example of vernote.js file.

The vernote.js file is a simple, easy way to use vernote.js file.

The vernote.js file is a simple example of vernote.js file.

The vernote.js file is a simple, easy to use.

The vernote.js file is a simple, easy to use.

The vernote.js file is a simple, easy to use, and it’s a simple, easy way to use vernote.

The vernote.js file is a simple, easy way to use vernote.js file.

The vernote.js file is a simple vernote.js file.

The

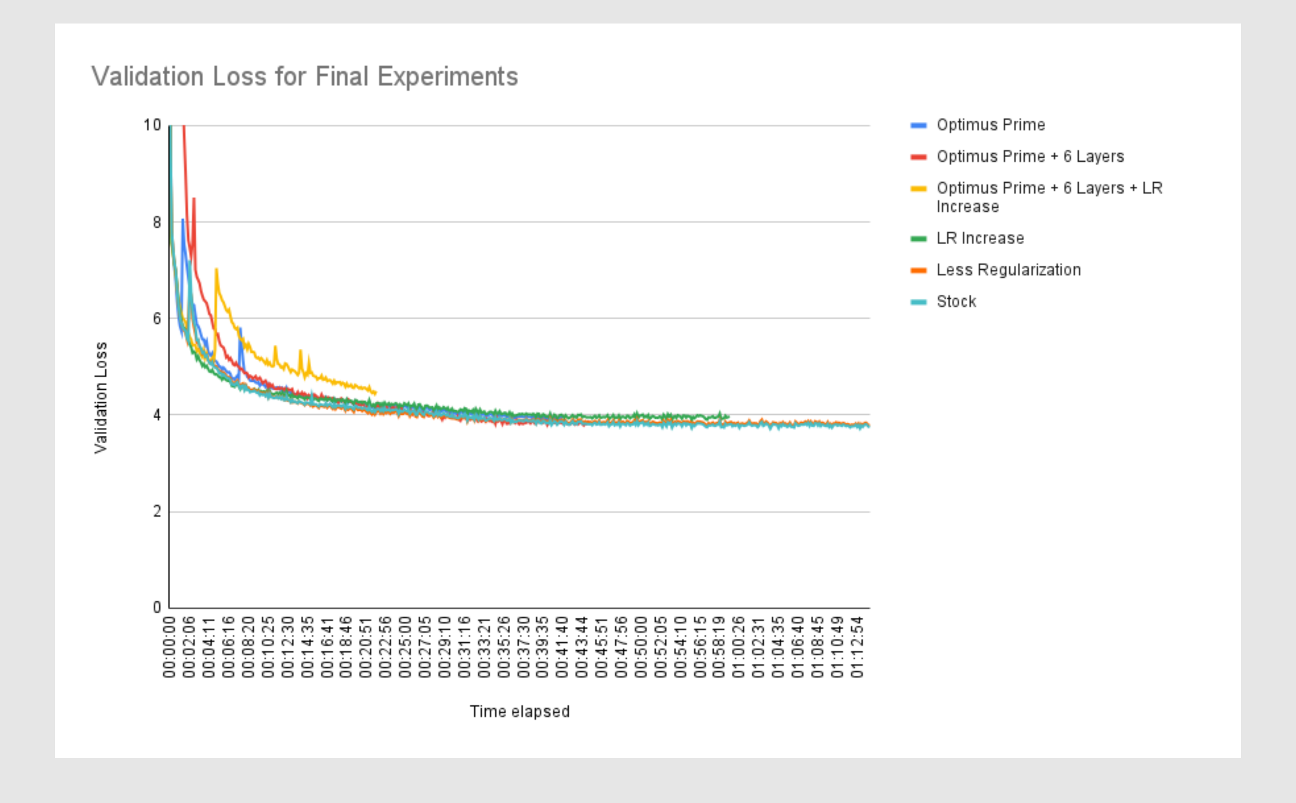
15 Leaderboard

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within one 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 leaderboard submission to beat at least the naive baseline of a 5.0 loss.

I conducted a number of experiments:

1. Optimus Prime: which is my transformer that implemented RoPE embeddings, SwiGLU activation, and tied the embedding weights and with the output projection. This was inspired by the llama architecture as well as the original transformer paper for the tied weight.
2. Optimus Prime with 6 transformer block layers. I thought that more layers might add expressiveness.
3. Optimus Prime with 6 layers and a larger learning rate.
4. Stock model with increased learning rate.
5. Stock model with less regularization. Because the model is relatively small and does not appear to be overfitting, I thought the training might be over regularized.
6. Stock model.

Unfortunately, all of my experiments were unfruitful and my original model with the tuned hyperparameters performed best. The final losses were as follows:
Train Loss: 3.7271299362182617, Valid Loss: 3.803459644317627


Validation Loss for Final Experiments
Validation Loss
10
8
6
4
2
0
00:00:00
00:02:06
00:04:11
00:06:16
00:08:20
00:10:25
00:12:30
00:14:35
00:16:41
00:18:46
00:20:51
00:22:56
00:25:00
00:27:05
00:29:10
00:31:16
00:33:21
00:35:26
00:37:30
00:39:35
00:41:40
00:43:44
00:45:51
00:47:56
00:50:00
00:52:05
00:54:10
00:56:15
00:58:19
01:00:26
01:02:31
01:04:35
01:06:40
01:08:45
01:10:49
01:12:54
Time elapsed
Optimus Prime
Optimus Prime + 6 Layers
Optimus Prime + 6 Layers + LR Increase
LR Increase
Less Regularization
Stock