# NanoGPT Speedrun Living Worklog

*How fast can I train GPT-2 on two RTX 4090 GPUs?*

March 8, 2025

I've seen some really awesome GPT-2 speedrun results from people like Keller Jordan, Fern, Braden Koszarsky, and others. I got a little inspired and wanted to see how fast I could train GPT-2 on my own hardware.

Technically, the NanoGPT speedrun is to train a neural network to 3.28 validation loss on FineWeb as fast as possible on an 8xH100 node. Keller Jordan maintains a leaderboard here. At the time of writing (Jan 16, 2025), the record is 3.14 minutes (!).

I have access to **2xRTX 4090 GPUs** and I want to see how fast I can train GPT-2 on them by following the same rules as the NanoGPT speedrun. If I see some success, I may try to transfer my methods to an 8xH100 node for comparison with the main leaderboard.

I'll be documenting my progress here and updating this post as I go. Code can be found in this GitHub repo.

## *Progress so far*

| # | Description | Record time | Training Tokens | Tokens/Second | Date | Commit | Log |
|---|---|---|---|---|---|---|---|
| 1 | Initial baseline | 8.13 hours | 6.44B | 221k | 2025/01/16 | b3c32f8 | here |
| 2.1 | Architectural changes | 7.51 hours | 5.07B | 188k | 2025/01/18 | b7bb93f | here |
| 2.2 | Muon optimizer | 4.53 hours | 3.04B | 187k | 2025/01/23 | b91c2c0 | here |
| 2.3 | Dataloading tweaks | 4.26 hours | 3.31B | 216k | 2025/02/18 | d59944d | here |
| 2.4 | Logit Soft-capping at 30 | 4.01 hours | 3.15B | 218k | 2025/02/23 | 12eab44 | here |
| 3 | Longer Sequence Length | 2.55 hours | 1.88B | 205k | 2025/03/03 | d982ed5 | here |

# *1. Initial setup and baseline*

Part of the goal of this project is for me to learn as I go, so I am going to start at the beginning - with with Andrej Karpathy's PyTorch GPT-2 trainer from llm.c. This is the script that Keller Jordan used for his initial baseline. This trainer is very similar to the NanoGPT trainer with some minor modifications / simplifications (such as no dropout).

I have upstreamed some QOL improvements and basic tweaks to the training script from Keller's fork, but have not changed any of the core training / modeling logic. Specifically:

1. Implemented gradient accumulation so that my 2x24GB GPUs simulate the training experience of a 8xH100 machine.
2. Increased learning rate to 0.0015 and halved the batch size (total batch size is 262144 - that is bs of `32/device * 2 devices * 1024 sequence length * 4 gradient accum steps`).
3. Improved learning rate schedule (linear warmup then linear decay).
4. Removed all affine scale/bias parameters and switched to RMSNorm.
5. Padded the vocab size from 50257 to 50304 to make it a multiple of 128 (for better tensor core utilization).
6. Using Pytorch 2.5.1 (the switch from 2.4 to 2.5 gave ~9% speedup on the 8xH100 leaderboard).

Additionally, I added `wandb` logging for easy tracking of training progress - optimistically I may need to remove this one day as it slightly increases step time.

Commit with the initial setup is here: `b3c32f8`.

The baseline run time on my 2xRTX 4090 setup is **8.13 hours**.

# *2. Implementing major improvements from the 8xH100 leaderboard*

Waiting 8 hours for a result is too slow for effective experimentation, so I'm going to begin by implementing some of the notable improvements from the 8xH100 leaderboard.

I'll start with the most impactful/easiest changes first:

1. Architectural changes and training tweaks
2. Muon optimizer
3. Dataloading tweaks
4. Logit Softcapping

## *2.1 Architectural changes and training tweaks*

There are some basic architectural changes and modernizations that can be made to the model that will speed up training. These changes are general improvements to the transformer decoder architecture that have been generally adopted since the original GPT-2 paper. The changes are:

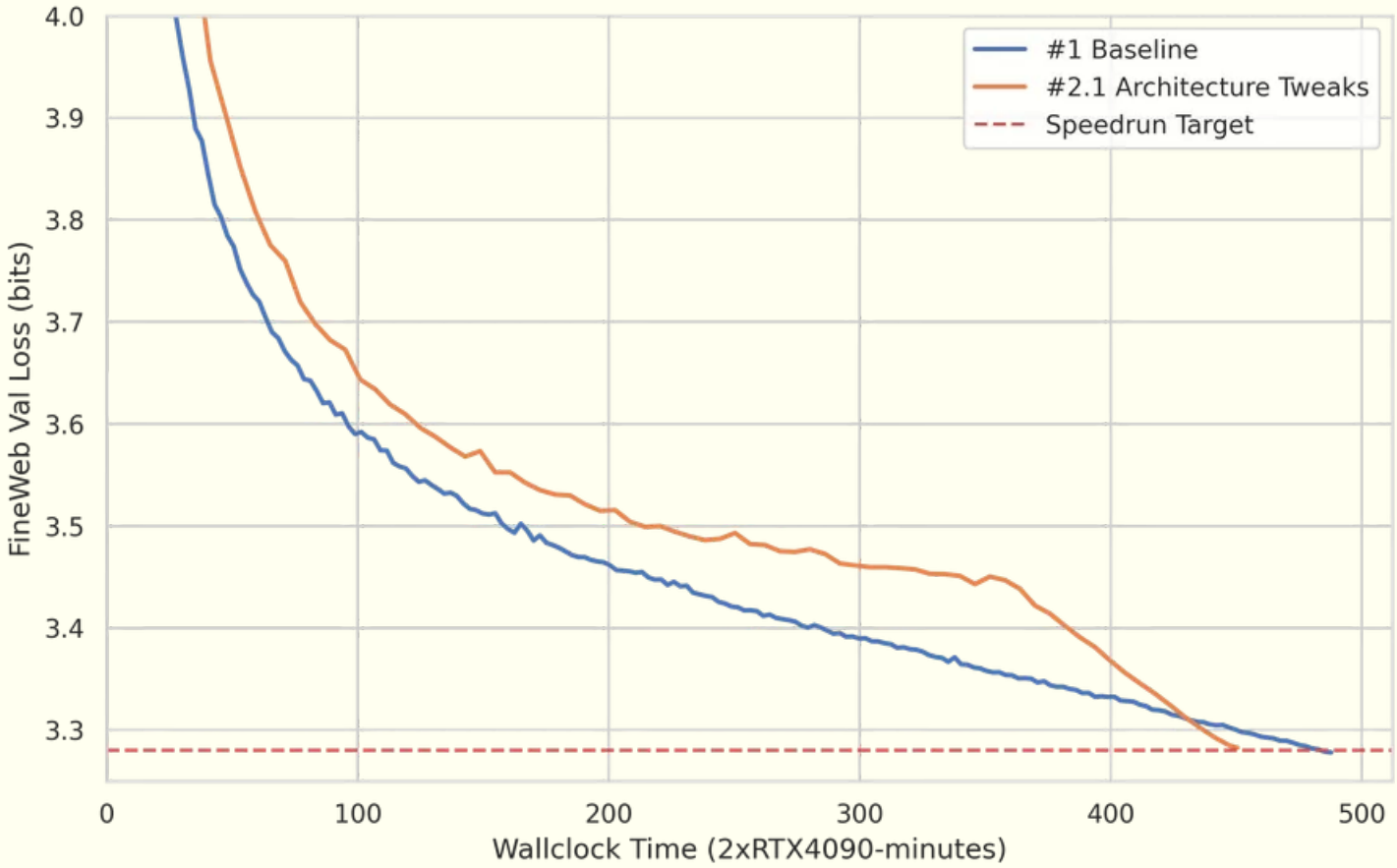
1. RoPE (Rotary Positional Embeddings). There are many good explanations of RoPE out there so I won't go into detail here.
2. ReLU^2 Activation[1]. Many activations that are better than GeLU have been proposed since GPT-2. ReLU^2 is a simple one that has been shown to be effective in decreasing training time required to reach a certain validation loss.
3. No gradient clipping. Gradient clipping can help stabilize training but it also slows down training. Since we are speed-running, we will remove gradient clipping. This also eliminates a hyperparameter that needs to be tuned.
4. Trapezoidal learning rate schedule. While cosine learning rate schedules are the de-facto standard, they can be difficult to work with since changing the number of training steps changes the entire schedule. Trapezoidal learning rate schedules are often easier to reason about / tune around, and they have been show to match the performance of cosine schedules.

In addition, learning rate and batch size have been tuned.

Once again, many of these changes are downstreamed from the modded-nanogpt repository / 8xH100 speedrun. Its not efficient to reinvent the wheel, and I want to get training time down as fast as possible in the beginning.

After implementing these changes (commit `b7bb93f`), the new run time is **7.51 hours**. This run was more data-efficient than the baseline, requiring only 5.07B tokens. However, the tokens/second increased, likely due to the larger batch size (more gradient accumulation

steps which tends to translate to lower throughput) and the architectural changes, such as the inclusion of RoPE. Once I have a shorter run time, I will be able to tune more effectively and see if I can remove gradient accumulation.



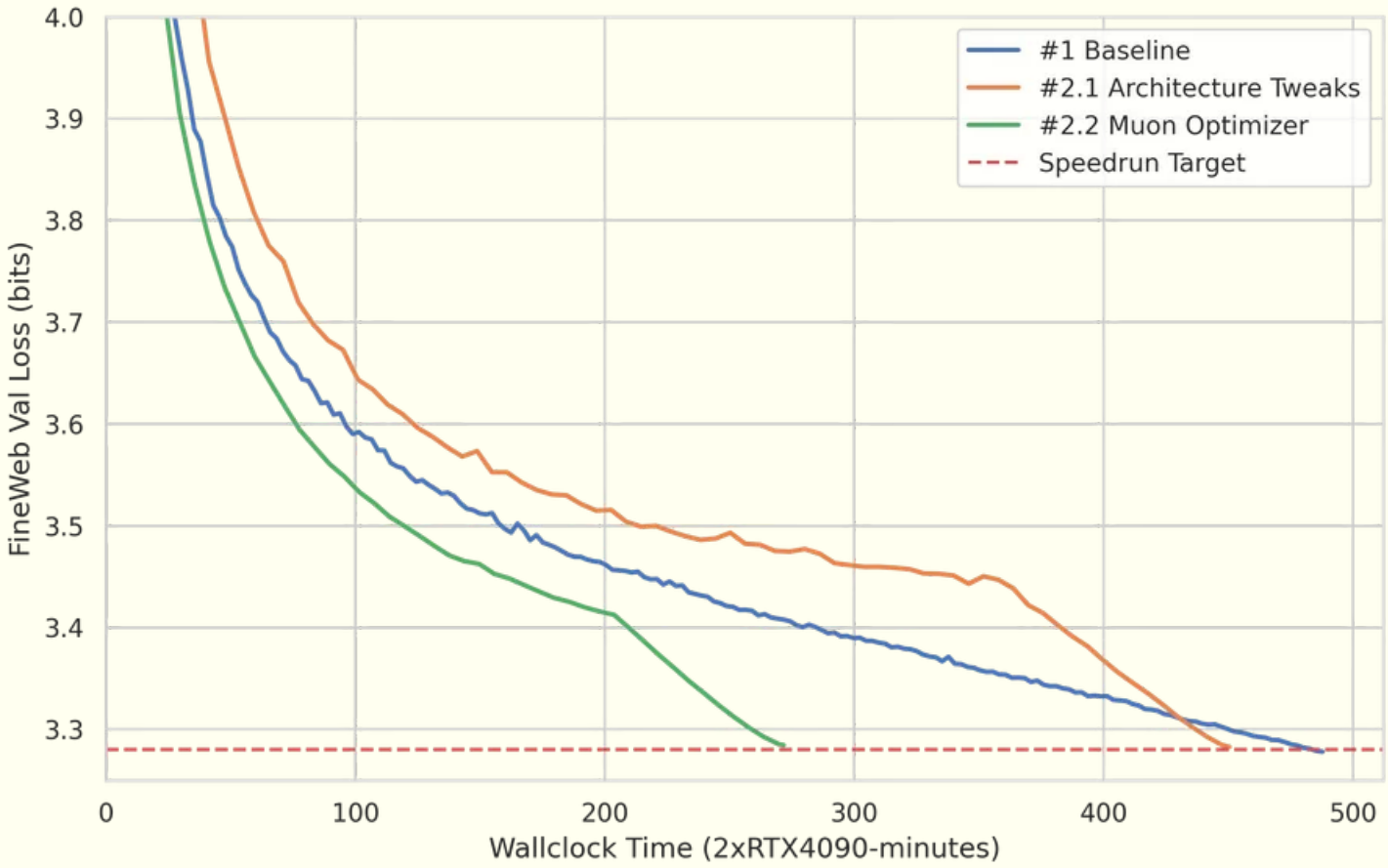

## *2.2 Muon Optimizer*

The Muon Optimizer is a new optimizer developed with and for the NanoGPT speedrun by Jordan et al. It is a variant of SGD with Momentum that applies a postprocessing step to the gradient updates to approximately orthogonalize each update matrix. Muon has some connections to approximate second-order optimizers[2] like Shampoo.

I highly recommend reading the original Muon blog post for more details, as well as checking out the optimizer comparison for GPT-2 speedrunning that Keller Jordan put to gether here. For those interested in a more step-by-step walkthrough of Muon, check out this excellent post by Jeremy Bernstein.

Muon is designed to work on *Linear* layers, so it is not quite a drop-in replacement for AdamW (e.g. it isn't meant to optimize Embedding layers). However it can be used to optimize all of the hidden layers of our GPT-2 model. The output `lm_head` layer and the token embeddings will still be optimized with AdamW.

Just like on the 8xH100 leaderboard, we observe a massive speedup when switching to Muon. The new run time is **4.53 hours**, requiring only 3.04B tokens. The tokens/second is also very similar to the previous run, which is a good sign that we are not losing throughput by switching optimizers.



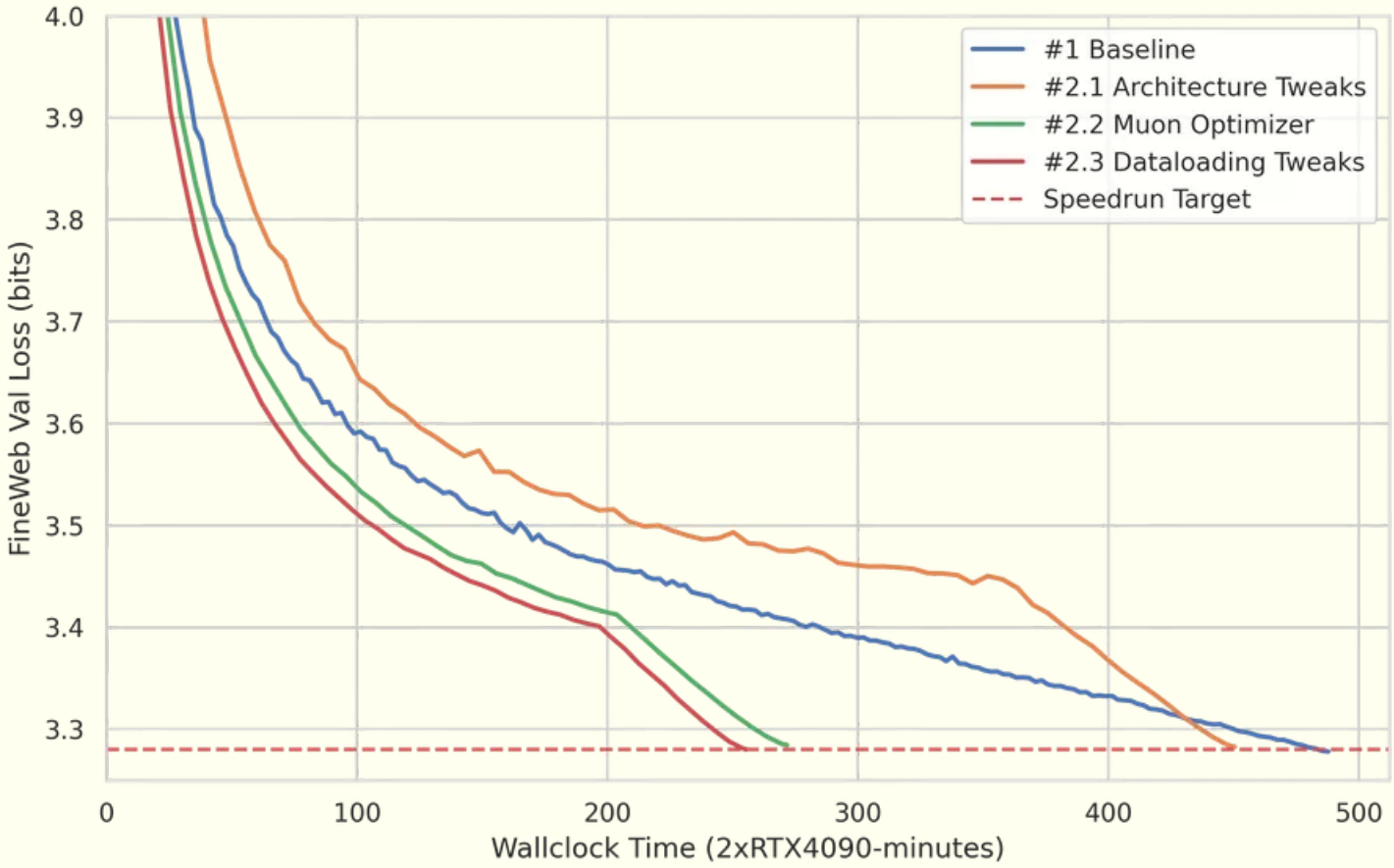

## *2.3 Dataloading Tweaks*

As we have improved our data efficiency via architecture tweaks and an optimizer change, our training throughput has dropped from 221k tokens/second to 187k tokens/second. That is a ~15% drop in throughput. Recovering most of that throughput could provide a significant improvement to our run time. An obvious place to start is with our dataloading and gradient accumulation logic.

Up until now, we have loaded a full-batch of data on each device and then split that full batch into smaller chunks (micro-batches) for each gradient accumulation step (recall that we are doing 8 accumulation steps per gradient update). We can instead make a minor tweak to our logic to load only the next micro-batch at each step of the dataloader, and then step the dataloader for each gradient accumulation step.

We also increase our torch version from `2.5` to `2.6` (which was recently released), and, in accordance with the new official rules designated on 2025/02/01, we have removed the use of `torch._inductor.config.coordinate_descent_tuning`.

These tweak brings our throughput back up to 216k tokens/second. In order to make runs more consistently hit the 3.28 validation loss target[3], we have also slightly increased the total number of training steps, so now 3.31B tokens are consumed. The new run time is **4.26 hours**, and the changes can be found at `d59944d`.




At this point, we code that can train GPT-2 almost twice as fast as the baseline.

## *2.4 Logit Soft-capping*

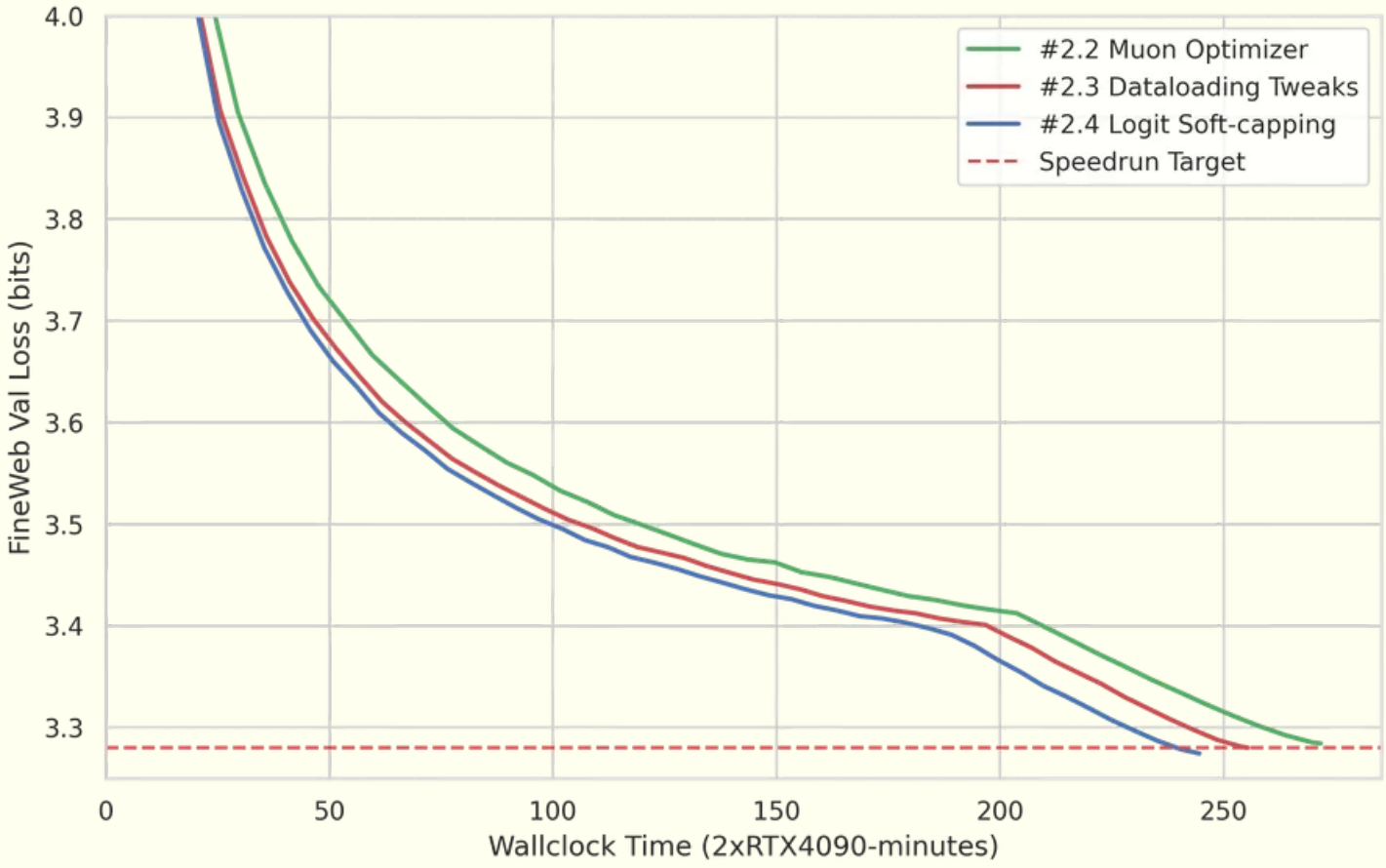
Logit soft-capping is a technique popularized by Gemma 2 and initially used to improve the NanoGPT speedrun by @Grad62304977.

Soft-capping is essentially a smooth and differentiable version of clipping⨁:

$$\text{softcap}(x, \text{cap}) = \text{cap} \cdot \tanh\left(\frac{x}{\text{cap}}\right)$$

Logit soft-capping prevents logits from growing excessively large by scaling them to a fixed range, which seems to help improve training dynamics. One could argue that this is imposing an inductive bias - and since we're in a relatively small model/low data regime that this is helpful.

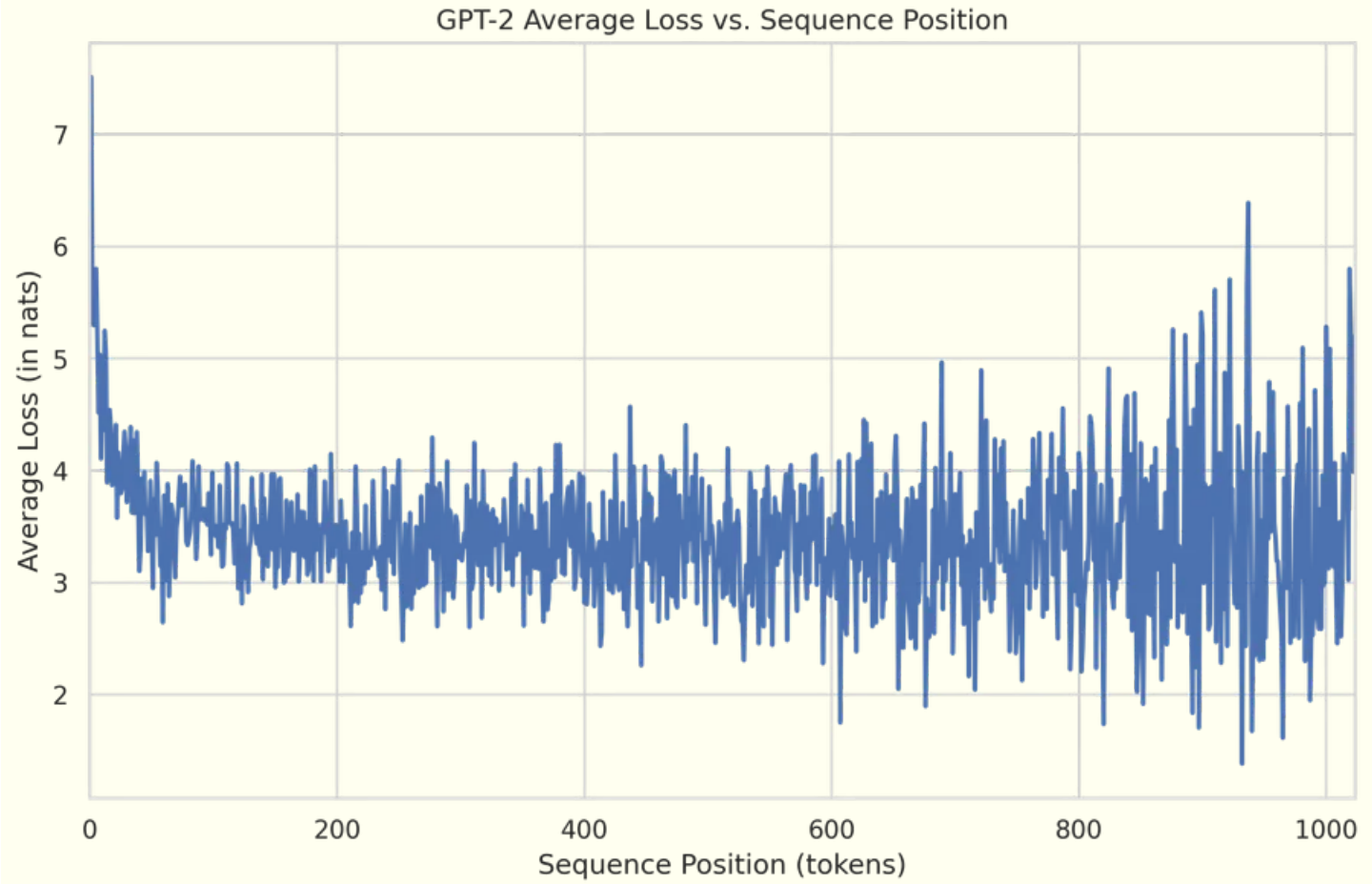
After implementing logit soft-capping with a cap of 30 (and doing some learning-rate tuning), the new run time is **4.01 hours**, requiring 3.15B tokens (commit `12eab44`). Throughput remained steady at ~218k tokens/second.




## *3 Longer Training and Evaluation Sequence Length*

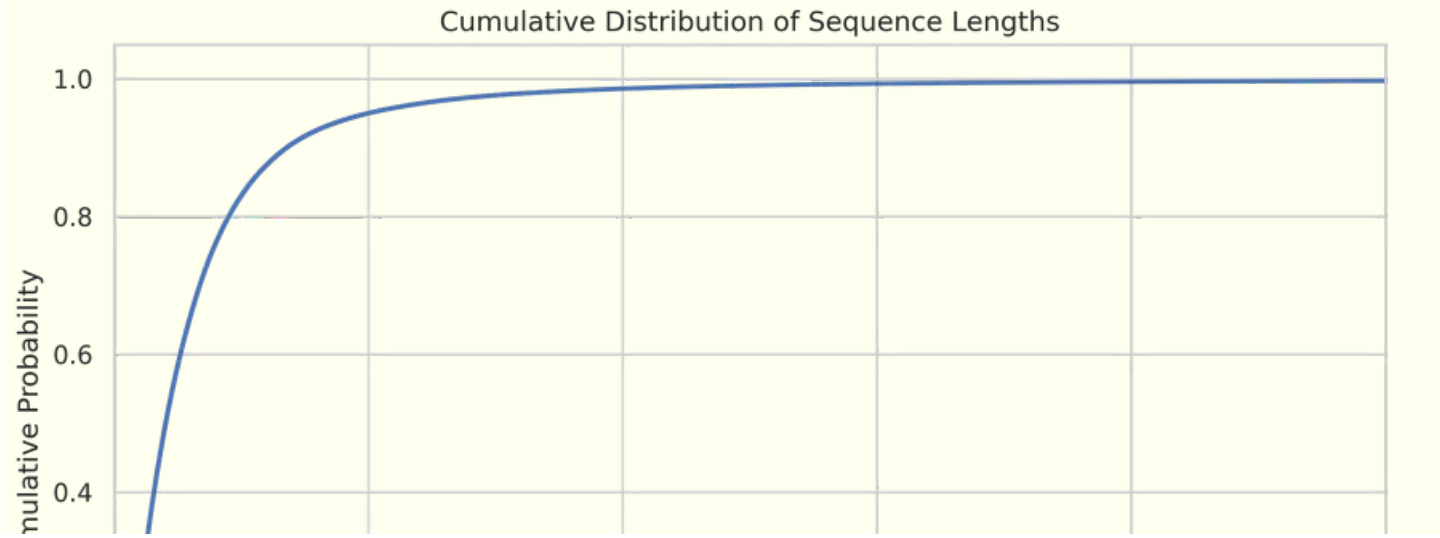
So far, we've been training and evaluating on sequences of 1024 tokens. We also haven't been particularly clever about how those sequences are processed. At each step, we simply load the next 1024 tokens into an element of the batch without regard for where the document starts or stops. That means much of the time we are starting in the middle of a document and cutting that document off before it reaches its end. We are also attending to tokens *across documents* since we're just using a simple causal mask.

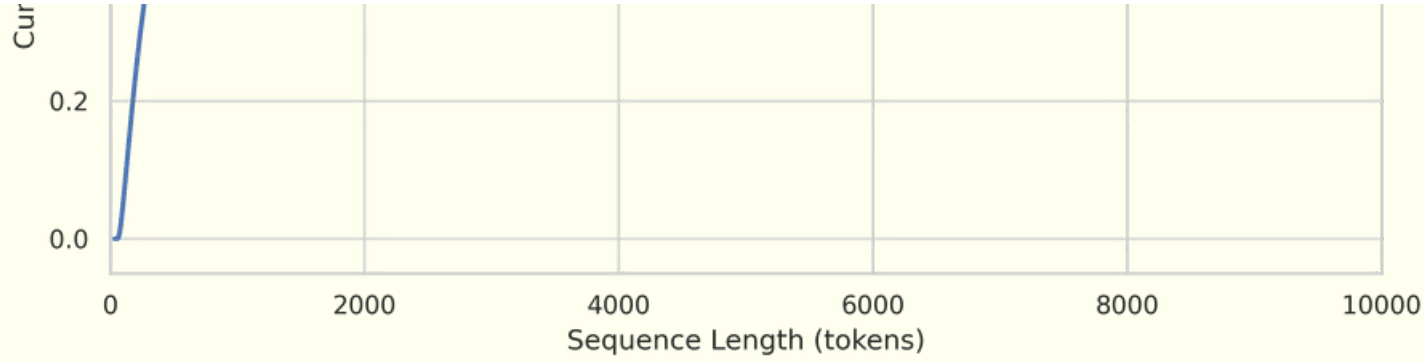
Cutting off documents in the middle is an especially large issue. See this plot of average loss vs sequence position:




Notice how the first twenty-five or so positions have a much higher average loss than the later positions. This is because at the beginning of the sequence the LLM has much less information with which to make informed predictions about the next token in the sequence. We want to avoid needlessly restarting documents/sequences in order to avoid this loss penalty!

A natural question to ask at this point is: how long are sequences in our dataset, on average?






The data reveals that approximately 20% of documents exceed our current 1024 token sequence length. By increasing the sequence length to >=8192 tokens, we can accommodate virtually all documents in our dataset without truncation.

To address the issues identified above, we'll implement two key improvements. First, we'll extend our sequence length to minimize document splitting across sequence boundaries. Taking this approach to its logical conclusion, we'll eliminate the traditional batch dimension entirely and instead maximize sequence length (effectively using a "batch size" of 1 that contains multiple concatenated documents). Second, we'll implement sophisticated attention masking that prevents cross-document attention while simultaneously leveraging the computational efficiency of sparse attention patterns.

Fortunately, FlexAttention provides an elegant solution that maintains the performance benefits of FlashAttention while enabling these improvements. One of FlexAttention's primary strengths is its ability to efficiently handle sparse, custom attention masks, making it ideal for our use case.

To implement FlexAttention, we need to define an appropriate attention mask that handles our specific requirements:

```
def make_attn_mask(idx, eot_token, window_size=1024):
    # Create a causal mask (only attend to past tokens)
    def causal_mask(b, h, q_idx, kv_idx):
        return q_idx >= kv_idx

    # Track document boundaries using end-of-text tokens
    documents = (idx == eot_token).cumsum(dim=1)

    # Only allow attention within the same document
    def document_mask(b, h, q_idx, kv_idx):
        return documents[b, q_idx] == documents[b, kv_idx]

    # Limit attention to an N-token window for efficiency
    def sliding_window_mask(b, h, q_idx, kv_idx):
        return q_idx - kv_idx <= window_size
```

```
    return and_masks(document_mask, causal_mask, sliding_window_mask)
```

Let's break down each mask:

1. **Causal Mask**: Standard in autoregressive language modeling. Ensures that tokens can only attend to previous tokens in the sequence, preventing information leakage from future tokens.

2. **Document Mask**: This restricts attention to tokens within the same document. By tracking document boundaries using end-of-text tokens, we prevent tokens from attending across different documents, which helps the model maintain coherent context within a single document.

3. **Sliding Window Mask**: This limits attention to a fixed window of tokens before the current position. This approach balances efficiency with context retention with a clear tradeoff: smaller windows are more efficient but may miss long-range dependencies, while larger windows capture more context at the expense of resources.

In order to build intuition about the individual component masks, we visualize them below:

When combined with the `and_masks` function, these three masks[4] work together to create an efficient attention pattern that respects document boundaries, maintains causality, and limits computational overhead for long sequences.

After incorporating FlexAttention with these masks, and increasing our sequence length to 32768 tokens, we observe a massive speedup[5]. The new run time is **2.55 hours**, requiring only 1.88B tokens (a huge data-efficiency improvement). Our throughput dropped slightly to ~205k tokens/second. See commit `d982ed5` for the full details.

# *References*

2024, Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977 (view online)

modded-nanogpt: Speedrunning the NanoGPT baseline

2024, Fern (view online)
hlb-gpt

2023, Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu (view online)
RoFormer: Enhanced Transformer with Rotary Position Embedding

2022, David R. So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V. Le (view online)
Primer: Searching for Efficient Transformers for Language Modeling

2024, Alexander Hägele, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro Von Werra, and Martin Jaggi (view online)
Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations

2022, Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre (view online)
Training Compute-Optimal Large Language Models

2024, Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein (view online)
Muon: An optimizer for hidden layers in neural networks

2018, Vineet Gupta, Tomer Koren, and Yoram Singer (view online)
Shampoo: Preconditioned Stochastic Tensor Optimization

2024, Jeremy Bernstein, and Laker Newhouse (view online)
Old Optimizer, New Norm: An Anthology

2024, Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju-yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev (view online)
Gemma 2: Improving Open Language Models at a Practical Size

2024, Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He (view online)
Flex Attention: A Programming Model for Generating Optimized Attention Kernels

2025, Jeremy Bernstein (view online)
Deriving Muon