# Practical Efficiency of Muon for Pretraining

**Essential AI**
San Francisco, CA
research@essential.ai

## Abstract

We demonstrate that Muon, the simplest instantiation of a second-order optimizer, explicitly expands the Pareto frontier over AdamW on the compute-time tradeoff. We find that Muon is more effective than AdamW in retaining data efficiency at large batch sizes, far beyond the so-called critical batch size, while remaining computationally efficient, thus enabling more economical training. We study the combination of Muon and the maximal update parameterization (muP) for efficient hyperparameter transfer and present a simple telescoping algorithm that accounts for all sources of error in muP while introducing only a modest overhead in resources. We validate our findings through extensive experiments with model sizes up to four billion parameters and ablations on the data distribution and architecture.

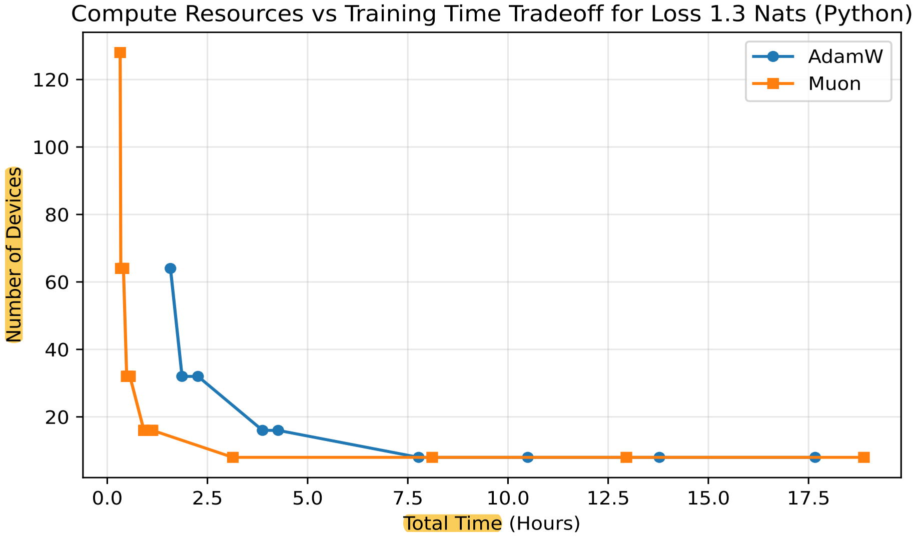

Figure 1: Muon (Jordan *et al.*, 2024) explicitly expands the Pareto frontier over AdamW on the compute-time tradeoff by retaining data efficiency at large batch sizes.

# 1 Introduction

Adam (Kingma and Ba, 2014) and AdamW (Loshchilov and Hutter, 2017) have empirically dominated the landscape of neural network optimizers in recent years. Superior generalization from the same training data (data-efficiency) at marginal increase in compute cost enabled them to gradually replace other first-order incumbents. By exploiting the statistical dependencies between model parameters, second-order optimizers have the potential to challenge the status quo. Recent second-order optimizers (Jordan *et al.*, 2024; Vyas *et al.*, 2024) indeed promise data-efficiency over AdamW for pretraining, without compromising the overall computational advantage. However, these efforts rely on wall-clock time (MLCommons, 2025) or FLOPs to abstract away from algorithmic complexity and implementation details. But, this is insufficient to conclude their advangage over AdamW. Pretraining workloads come in various model shapes, data sizes, and resource affordances. To fully explain the practicality of an optimizer one needs to capture the fundamental tradeoff between compute and time resources—the ability to reduce total training time by using more devices. Since this trade-off is what makes large-scale pretraining feasible, it is crucial to characterize how it changes when comparing optimizers for pretraining.

A key element in the compute-time tradeoff is the batch size. We may process data faster by increasing the batch size (i.e., by increasing the number of devices in distributed training) but likely at the cost of data efficiency - the loss reduction the model achieves per token. A "good" optimizer should therefore support large batch sizes to facilitate faster training while remaining persistently data-efficient. Simultaneously, it must remain computationally efficient by maintaining or improving FLOP efficiency and hardware utilization, ideally with any reasonable implementation. How can we take into account all these considerations to show that a second-order optimizer yields a more favorable compute-time tradeoff over AdamW?

To answer this question, we directly plot the compute and time resources required to achieve the same target loss using different optimizers. On this two-dimensional plane, optimizers are represented as iso-loss frontiers that marginalize over all other variables affecting the final tradeoff, thus admitting effective comparison. To study the tradeoff, we focus on Muon (Jordan *et al.*, 2024), the simplest instantiation of a second-order optimizer. We build on the recent work by Liu *et al.* (2025), which shows that Muon is more FLOP-efficient than AdamW, and make the stronger claim that Muon expands the Pareto frontier on the compute-time tradeoff with variable batch sizes, thereby strictly increasing the practitioner's flexibility in resource allocation (Figure 1). To analyze this behavior, we present a study of the relative data efficiency of Muon over AdamW in the large batch size regime.

In the second part of the paper, we address another core challenge in optimization for pretraining: how to choose optimal hyperparameters. We focus on the maximal update parameterization (muP) (Yang *et al.*, 2022), a principled approach to hyperparameter transfer. It stipulates certain weight initialization and learning rate scaling such that optimal hyperparameters identified on a small model remain effective on a large one, reducing the burden of hyperparameter search at a large model scale. We demonstrate successful hyperparameter transfer with muP under Muon, extending known results under AdamW. While muP is asymptotically correct in the limit on the network width and search granularity, in practice it suffers from estimation errors and requires multi-scale tuning over different model sizes. We develop a simple "telescoping" algorithm for training a final model of cost $C$ and width $N$ that adjusts the search grid across model scales to account for these estimation errors with an additional modest $O(C \log{(N)})$ compute cost (Figure 6).

Our paper serves as a practitioner's guide on optimization for pretraining. Our final recommendation is to choose Muon over AdamW because it increases flexibility in resource allocation by remaining data-efficient with large batch sizes, and to combine it with muP for compute-efficient hyperparameter search at scale. We back our findings through extensive experiments, varying the model size up to 4 billion parameters and batch size up to 16 million tokens, ablating the effect of the data distribution and architecture.[1]

[1] All our experimental artifacts will be released at: `https://huggingface.co/EssentialAI`.

# 2 Muon Improves the Compute-Time Tradeoff

## 2.1 Review of Muon

A transformer language model primarily consists of matrix weights $W \in \mathbb{R}^{m \times n}$ parameterizing the feedforward and attention layers. We assume $m \leq n$ without loss of generality. A weight update takes the form of $W_{t+1} = W_t - \eta_t O_t$ where $O_t \in \mathbb{R}^{m \times n}$ is some transformation of the gradient $G_t \in \mathbb{R}^{m \times n}$ on the $t$-th batch. Muon can be seen as matrix-structured steepest descent with spectral norm regularization:

$$O_t = \operatorname*{arg\,min}_{O \in \mathbb{R}^{m \times n}:\ ||O||_2 \leq 1} \operatorname{tr}\left(G_t^\top O\right)$$

It is easy to show that $O_t = UV^\top$ is an optimal solution where $G_t = U\Sigma V^\top$ is a singular value decomposition (SVD) (Bernstein and Newhouse, 2024). Muon avoids explicit computation of SVD by Newton-Schulz iteration (Kovarik, 1970; Björck and Bowie, 1971). In practice, we combine Muon with Nesterov momentum, learning rate scaling, and (coupled) weight decay. Thus the only state Muon maintains is the first moment $M_t \in \mathbb{R}^{m \times n}$ initialized to zero. In the $t$-th update, given the gradient $G_t$ we compute

$$\begin{aligned} M_t &= G_t + \beta M_{t-1} \\ O_t &= \textbf{NewtonSchulz}(G_t + \beta M_t) \\ W_{t+1} &= W_t - \eta_t((0.2\sqrt{n})O_t + \lambda W_t) \end{aligned} \tag{1}$$

where $\beta, \lambda \in \mathbb{R}$ are momentum and weight decay hyperparameters. The constant $0.2\sqrt{n}$, proposed by Liu *et al.* (2025), normalizes the Muon update to have a similar RMS value as the AdamW update.[2] This ensures that the same learning rate schedule and weight decay work well for both Muon and AdamW, which is useful because AdamW is used to update embedding and normalization parameters. Jordan *et al.* (2024) provide well-tuned hyperparameters for Newton-Schulz that rapidly flatten all singular values to range (0.7, 1.3).

### 2.1.1 Why we focus on Muon

While Muon is not the only viable second-order optimizer, we limit our scope to Muon for the following reasons. First, it has the simplest derivation and implementation, allowing us to focus on characterizing the practical benefits of second-order methods. Second, we can reduce more sophisticated algorithms such as Shampoo (Gupta *et al.*, 2018; Anil *et al.*, 2020) and Soap (Vyas *et al.*, 2024) directly to Muon under simplifying assumptions (Appendix B). Thus Muon is in some sense the minimal version of this class of optimizers. Third, Muon has the lightest memory footprint of all the optimizers we considered (even lighter than AdamW) since it only maintains the first moment, reducing the infrastructure work needed for distributed training. Fourth, the FLOP overhead of Muon over AdamW diminishes in the batch size $B$ as $\Theta(\frac{m}{B})$, making Muon a promising candidate for large-batch training. Fifth, given the default hyperparameters in Newton-Schulz iteration, Muon only needs tuning over the maximum learning rate and weight decay, which are furthermore compatible with AdamW under the constant $0.2\sqrt{n}$ rescaling (Liu *et al.* (2025)) and make comparison with AdamW especially convenient.

## 2.2 Experimental Setup

**Architectures.** We use a modern decoder-only transformer model based on Gemma 3 (Gemma Team, 2025), which replaces the soft-capping method in Gemma 2 with QK-norm for attention stability. Unlike Gemma 3 which interleaves local and global attention layers, we only use global attention layers for simplicity. We compare Gemma 3 with Gemma 2 in our ablation studies.

**Data distributions.** A typical pretraining data mix consists of many different sources (GitHub code, math blogs, books, arXiv articles, general web, etc.). We use data distributions with markedly different compression ratios and target capabilities, to reflect the heterogeneity of the pretraining

[2] 0.2 is the empirically observed RMS of a typical AdamW update. $1/\sqrt{n}$ is the RMS of a perfectly normalized $O_t$, which follows immediately from the fact that the Frobinius norm is the sum of singular values.

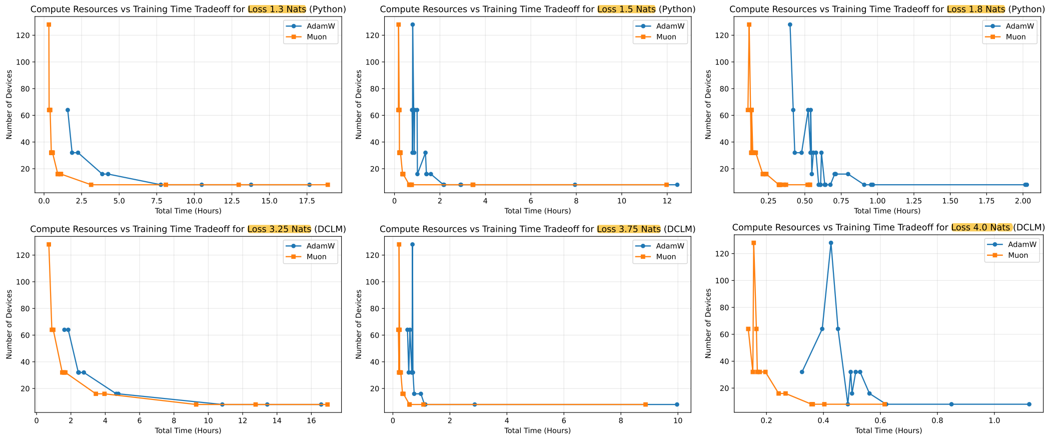

Figure 2: Muon expands the Pareto frontier over AdamW on the compute-time tradeoff at various loss thresholds on Python (top) and DCLM (bottom).

mix. Specifically, we focus on text and code. For text, we use high-quality general web text from DCLM (Li *et al.*, 2024). For code, we use Python code obtained from our in-house filtered version of the Stack V2 (Lozhkov *et al.*, 2024). We chunked input sequences to length 8192 using the Llama3 tokenizer with the vocab size of 128K.

**Token budget.** To ensure training is converged, we make an effort to use enough tokens to meet or exceed the Chinchilla-optimal token budget under our resource constraints (Hoffmann *et al.*, 2022). Specifically, we trained the 100M model size on 10B tokens (5×), 500M on 25B tokens (2×), and each of the 1B, 2B, and 4B model sizes on 50B tokens (2.5×, 1.25×, and 0.625×).

**Optimizer implementations.** We use the AdamW implementation (`optax.adamw`) in the Optax library. We use a straightforward in-house Jax implementation of Muon described in Section 2.1, applying (1) to all layers except embedding and normalization (Appendix G). For the latter layers, we use Optax Adam with the same learning rate and weight decay while holding $\beta_1 = \beta_2 = 0.95$ fixed. We train our models on TPU v5p chips, achieving close to 50% model FLOP utilization (MFU).

**Hyperparameter tuning.** We ran a fine-grained sweep at the 100M model scale, then validated the efficacy of the chosen hyperparameters at the 500M scale. For the 1B model, we ran a smaller sweep across different batch sizes, and then carried those findings to the 2B and 4B experiments. In all runs, we use the learning rate schedule of a linear warmup and cosine decay to 0.1 of the max learning rate.

**Initial verification of Muon's performance** We were able to quickly obtain positive results with the above setup (Appendix C). Specifically, we confirmed that at any given number of steps, Muon consistently achieved a lower training loss.[3] We also confirmed that Muon achieves a target loss faster than AdamW in wall time.

### 2.3 The Compute-Time Tradeoff for Pretraining

Conventional approaches that simply compare optimizers in wall time or FLOP efficiency do so at fixed resources (i.e., devices) and fail to characterize the tradeoff between resources and training time. Instead, we compare optimizers as iso-loss curves on the compute-time plane, by measuring the total training time to reach a target loss as a function of the number of devices and correspondingly the batch size. The gap between the curves represents novel economically feasible options (e.g., shorter run or fewer devices than what is possible under the other optimizer).

[3] Since we do not epoch over data, we use the training loss on a new batch as a proxy for generalization error for convenience. While this is limited in quantity compared to using a separate held-out dataset, it achieves the same effect when averaged over many batches.

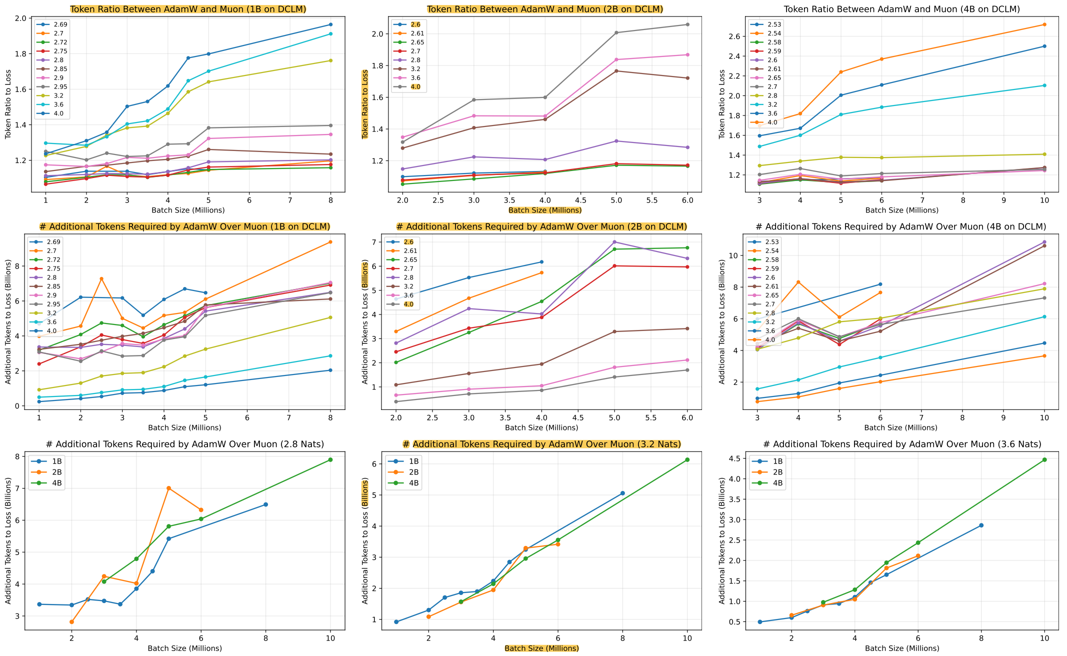

Figure 3: Relative data efficiency of Muon over AdamW. (Top) Token ratio vs batch size plots for different target losses across model sizes. (Middle) Corresponding token difference plots. (Bottom) Token difference across model sizes at fixed target losses

We simulate how a practitioner may use more devices to reduce the total training time by training 500M parameter models over 13 batch sizes ranging from 128K to 16M with data parallelism, where we vary the number of devices from 8 to 128 in a realistic way to ensure enough on-device memory.[4] We show the resulting plots in Figure 2. The plot shows that Muon explicitly expands the Pareto frontier on the compute-time tradeoff, thereby strictly increasing the practitioner's flexibility in resource allocation. The expansion is consistent on different thresholds and data distributions. See Appendix E for overlaid thresholds.

The plot gives a concise summary of Muon's practical efficiency over AdamW, marginalizing over all variables that affect the final compute-time tradeoff in complex and possibly competing ways. For instance, increasing the batch size may increase the per-device MFU (assuming enough devices) and thereby increase the number of FLOP/s, but it may simultaneously decrease the data efficiency of the optimizer and thereby require more FLOPs to reach the same loss. The plot also abstracts away implementation details of the optimizers, which may raise the concern that the gain unfairly comes from AdamW's suboptimal implementation. However, we use the standard Optax AdamW implementation which we view as close to optimal. On the other hand, our Muon implementation is quite naive and thus likely to only strengthen the efficiency gain with further optimization.

## 2.4 Muon's Relative Data Efficiency Over AdamW

Misspeak?

As we add more devices with data parallelism, we need to increase the batch size correspondingly to ensure that the devices are not memory-bound. At the same time, the data efficiency of an optimizer begins to diminish once the batch size passes a certain threshold, requiring more tokens to reach the same loss. Thus we hypothesize that Muon improves the compute-time tradeoff by remaining more data-efficient than AdamW at large batch sizes.

[4] Specifically, we use 8 devices for batch size under 1M, 16 for under 2M, 32 for under 4M, 64 for under 8M, and 128 for above 8M.

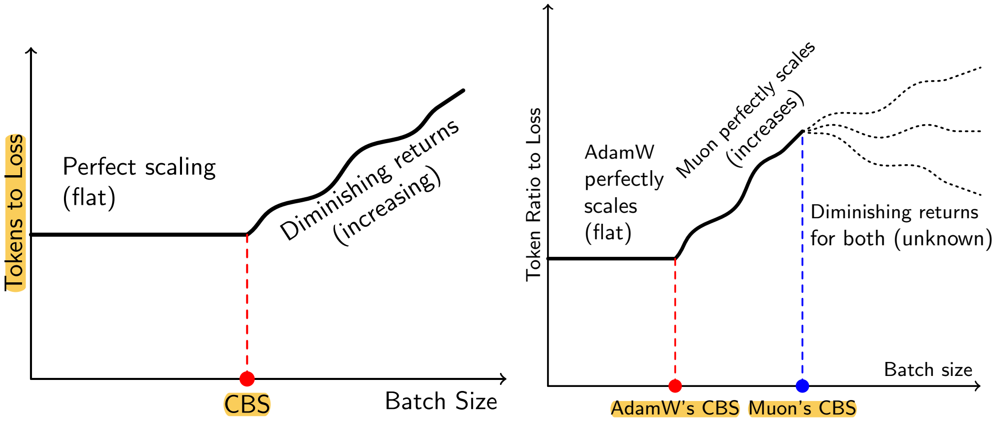

Figure 4: *Token ratios to loss* give a clearer picture of the practical advantages of Muon over AdamW, compared to *tokens to loss*

To characterize the relative data efficiency of Muon over AdamW, we propose measuring the ratio of their token consumptions:

$$R_L(B) = \frac{T_{L,A}(B)}{T_{L,M}(B)} = 1 + \frac{T_{L,A}(B) - T_{L,M}(B)}{T_{L,M}(B)} \tag{2}$$

where $T_{L,A}(B)$ is the number of tokens consumed to reach a target loss $L$ at batch size $B$ under AdamW, similarly $T_{L,M}(B)$ for Muon. The ratio represents data overhead: how many more tokens does AdamW need to reach the same loss, relative to Muon's own token consumption? If $R_L(B)$ remains above 1, and is increasing or at least nondecreasing when $B$ is large, the relative data efficiency of Muon over AdamW does not vanish even in the over-large batch size regime.[5]

Figure 3 (top row) shows empirical observations with 1B model size on DCLM. We see that the token ratio is above 1 and generally increasing at higher losses and at least constant at lower losses, suggesting that Muon's relative batch size benefit persists beyond any fixed batch sizes. It also implies that the unnormalized additional number of tokens $T_{L,A}(B) - T_{L,M}(B)$ strictly increases in $B$ (middle row). We observe that the number of extra tokens required by a model is surprisingly agnostic to model size (bottom row). For instance, to reach a target loss of 3.2 nats, all three of the 1B, 2B and 4B model need around 2B extra tokens with AdamW than Muon.

The empirical fact that $R_L(B) > 1$ is nondecreasing means the non-optimizer FLOP overhead of AdamW stays constant even at large batch sizes. This gives Muon an edge in trading off compute resources (i.e., more devices with bigger batch sizes) and training time, despite the fact that Muon performs more FLOPs than AdamW.

### 2.5 Connection to Critical Batch Size

A great deal of research on the impact of batch size on training focuses on the idea of "critical" batch size, which is broadly defined as the largest batch size that allows for almost linear or "perfect" scaling (e.g., doubling the batch size halves the number of steps) beyond which we receive diminishing returns (e.g. Zhang *et al.*, 2019). We can relate the token ratio $R_L(B)$ to the critical batch size as follows. We can define the critical batch size as a value $B^\star_L$ such that (1) the number of tokens required to reach the loss remains constant $T_L(B) = T^\star_L$ for batch sizes $B \leq B^\star_L$, and (2) $T_L(B) = \mathrm{inc}_L(B)$ is some increasing function for $B > B^\star_L$. Even if Muon's critical batch size is larger than Adam's, $R_L(B)$ may either increase, decrease, or stay constant depending on the tail behavior of token consumption under the two optimizers (Figure 4). Our empirical finding is that $R_L(B)$ is nondecreasing (Figure 3), providing information on the post-critical batch size regime. Appendix D gives more detailed analysis.

[5] We note that $R_L(B)$ translates to non-optimizer FLOP overhead in training, assuming the standard $6dT$ FLOP count approximation for training a transformer language model with $d$ parameters on $T$ tokens.

## 3 Choosing Hyperparameters for Muon

Up until this point, we have discussed the benefits of replacing AdamW with Muon in pretraining. To utilize an optimizer efficiently, it is also important to be able to calibrate it efficiently—specifically to be able to select near-optimal hyperparameters. Moonshot (2025) and Liu *et al.* (2025) leave open the question of the compatibility of Muon with standard hyperparameter transfer techniques, such as muP. Hence, after introducing the sources of error in muP in Sec. 3.2, in Sec. 3.4 we give the first empirical demonstration of muP used to calibrate hyperparameters for a large language model using Muon as the optimizer. We note that while Wang and Aitchison (2024) considers the hyperparameter transfer of decoupled weight decay, we consider the coupled setting and hence inherit the transfer properties discussed in Yang *et al.* (2022). First, in the next section, we introduce the high-level ideas of muP.

### 3.1 The Maximal Update Parameterization (muP)

Brute-force grid searches over hyperparameters become intractable for large models. For $k$ hyperparameters each on an $m$-point grid, we must train $m^k$ separate models, which can be prohibitively expensive. As an example, the largest model trained in this section has 3.7B parameters and took 100 hours on 128 TPU v5ps to process 160B tokens; a modest $8 \times 8$ grid search on the full model would take nearly 820,000 device hours. Comparatively, Liu *et al.* (2024) trained a 671B model with 37B active parameters with 14.8T tokens and required only 2.788M H800 GPU hours, so such a grid search would consume nearly 30% of the total compute budget while training a substantially smaller model.

Hyperparameter transfer is a practical alternative: train a smaller "proxy" model, tune its hyperparameters, and apply the resulting settings to the larger model. However, without further insight, there is no guarantee that the optimal hyperparameters for the smaller model work well on the larger one. MuP (Yang *et al.*, 2022; Yaida, 2022) solves exactly this issue, ensuring consistent and predictive hyperparameter transfer across model scales. We introduce the core motivating discussion in App. I and provide explicitly our choice of initialization, multiplier and learning rate scaling in Table 5.

Specifically, we define a width-dependent parameterization of the network using three width-dependent functions $a(n), b(n)$ and $c(n)$. Each weight matrix $W_\ell$ at layer $\ell$ is rescaled as $W_\ell = a(n)w_\ell$, where $w_\ell$ are the trainable parameters, each entry of $w_\ell$ is initialized as $w_\ell \sim \mathcal{N}(0, b(n))$ and the learning rate is set to $c(n)\eta_0$ for some width-independent base rate $\eta_0$.

| | **Input Weights & Biases** | **Output Weights** | **Hidden Weights** |
|---|---|---|---|
| **Multiplier** $a(n)$ | $\sqrt{\mathrm{fan\_out}}$ | $\frac{1}{\sqrt{\mathrm{fan\_in}}}$ | $1$ |
| **Initialization Variance** $b(n)$ | $\frac{1}{\mathrm{fan\_out}}$ | $\frac{1}{\mathrm{fan\_in}}$ | $\frac{1}{\mathrm{fan\_in}}$ |
| **Learning Rate** $c(n)$ | $\frac{1}{\sqrt{\mathrm{fan\_out}}}$ | $\frac{1}{\sqrt{\mathrm{fan\_in}}}$ | $\frac{1}{\mathrm{fan\_in}}$ |

Figure 5: muP hyperparameter scaling rules, reproduced from Yang *et al.* (2022). Here, fan_in and fan_out denote the input and output dimensions, respectively, for a given weight matrix, e.g. for the weight matrix in the $\ell^{\text{th}}$ layer, $W_\ell \in \mathbb{R}^{m \times n}$ fan_in is $n$ and fan_out is $m$.

### 3.2 Sources of Error

This section analyzes the dominant sources of error in hyperparameter transfer under muP, introduced in the previous section. In the next section (Sec. 3.3), we will introduce a simple algorithm for controlling and suppressing the errors discussed in this section. The core analysis we present, depends on the well-known fact that neural networks often admit well-defined infinite width limits (e.g. Hanin (2023)). That is, given a large language model with width parameters (e.g. number of heads, hidden dimension and MLP dimension) there is a family of large language models $f(x, n)$ for hyperparameter $x$, that we can construct by increasing the width $n$. The core assumption we will make is that this limit exists, call it $f_0(x)$, and is smooth in $n$, the network width. Choosing a simple re-parameterizization of the network in $1/n$ (which is small, for large $n$) instead of $n$, we can apply a Taylor expansion to $f(x, 1/n)$ as

$$f(x, 1/n) = f_0(x) + \frac{f_1(x)}{n} + O\left(\frac{1}{n^2}\right). \tag{3}$$

Imprecision from finite width

(The full analysis is more complicated, as discussed in Yang (2019) and Yaida (2022), and in general we need to ensure that the infinite series can be truncated at finite order. For our setting, however, this is sufficient.) From this analysis alone, it is clear that there is imprecision introduced in using any finite width, $n$, for hyperparameter transfer. The theory introduced in Yang (2019) and experimentally verified in Yang *et al.* (2022) enables principled transfer of hyperparameters from small to large models by aligning their training dynamics across different model widths. However, numerous recent experiments (e.g., in the appendix of Everett *et al.* (2024)) demonstrate that such transfer is only approximately valid even at relatively large widths, with errors that diminish as the model becomes wider. This observation aligns with the central tenet of muP: hyperparameter transfer becomes exact only in the infinite-width limit.

We identify two primary sources of error in the muP hyperparameter transfer protocol. First, for a loss function $\mathcal{L}$, the optimizing hyperparameter $x^*(n)$ of $\mathcal{L}(f(x, n))$ at width $n$ deviates from the optimizing hyperparameter of its infinite width counterpart $x^*$ that optimizes $\mathcal{L}(f_0(x))$ due to $1/n$-order corrections. It follows from Eq. 3 that the optimal hyperparameter $x^*(n)$ minimizing the loss shifts as

$$\underset{\text{hyperparam of finite width model}}{x^*(n)} = \overset{\text{hyperparam of infinite width model}}{x^*} \underset{\text{error}}{- \frac{\alpha}{n}} + O\left(\frac{1}{n^2}\right), \tag{4}$$

for some constant $\alpha$ depending on the network and loss function (see App. J for the specific details). This finite-width bias represents an unavoidable source of error unless the proxy model is sufficiently large.

Second, even for stationary loss minima (i.e. $x^*(n) = x^*$), mesh-based approximations during the hyperparameter sweep introduce sampling error. If $\hat{x}^*(n)$ is the discrete optimum estimated by evaluating a grid of hyperparameters on a model of width $n$, then

$$\hat{x}^*(n) \approx x^*(n) + \varepsilon, \tag{5}$$

choosing too sparse values for HPs to sweep over

where $\varepsilon$ reflects the resolution of the sweep. In practice, coarse meshes, limited budget, or poorly chosen sweep ranges may lead to $\varepsilon$ large enough to obscure critical features of the loss landscape. In the next section, we introduce an algorithm to control for both sources of error independently, with nearly optimal complexity.

### 3.3 The "Telescoping" Protocol

Algorithm 1 presents a practical method for systematically controlling these errors. The intution behind our "telescoping" algorithm is that because we have the sources of error mentioned in Sec. 3.2, we train models at a range of widths, but also reduce the number of grid points at the larger widths due to their increased cost. The idea of intelligently constraining and allocating resources for hyperparameter search is not new (Fetterman *et al.*, 2023; Li *et al.*, 2018), and our idea of logarithmically reducing the search space more generally fits in the category of hierarchical Bayesian optimization (Kennedy and O'Hagan, 2000).

The accompanying parameterized analysis (Fig. 13 in the Appendix) shows that, depending on the desired confidence level, the compute budget for the final model training may vary between 20% and 99% of the total training and hyperparameter tuning cost; the compute saved when compared to a brute force grid search is typically higher than 50%. As a result, the total savings in large-scale training remains substantial.

**Algorithm 1:** Telescoping Algorithm for Hierarchical Hyperparameter Transfer

```
Input: Base model width $N_0$, final calibration width $N_c$, final model width $N$, number of
       hyperparameters $k$, number of sweep points $m$
Output: Sequence of optimal hyperparameters with controlled drift up to width $N$
Initialize: Determine mesh size using a curvature estimate near the optimum at $N_0$. Set $n \leftarrow N_0$;
Perform hyperparameter grid sweep at width $N_0$ with full mesh;
while $n < N_c$ do
    $n \leftarrow 2n$ ;                // Double the model width (Increases cost by 4×)
    Reduce number of sweep points per hyperparameter by factor $4^{-1/k}$;
    Adjust mesh resolution accordingly (submesh of size $4^{-1/k}$);
    Perform the grid sweep over the refined mesh;
```

The core assumption behind the telescoping approach presented in Alg. 1 is that the optimal hyperparameter varies smoothly with model width, with a shift of order $1/n$. Under this assumption, when the width doubles (which increases the FLOPs by roughly a factor of four), halving the mesh spacing suffices to track the $1/n$ drift while keeping the total number of samples of each hyperparameter at each stage proportional to $4^{-1/k}$. Thus, the cost of each successive stage is roughly constant. Crucially, this prevents *under*-refinement (missing the shifted minimum) and *over*-refinement (unnecessary fine-grained sweeps).

A sufficiently broad initial sweep at the smallest width $N_0$ usually leverages prior empirical knowledge about feasible hyperparameter ranges. As $n$ doubles, the new sweep narrows the mesh just enough to capture any incremental drift of the optimum. Absent any new peaks emerging at later widths—which empirical practice suggests is relatively rare—this "half-meshing" strategy is near-optimal. It balances the competing needs of accuracy in locating the optimal hyperparameter and computational efficiency in the sweep. Because each refinement stage maintains approximately constant cost (training is more expensive but the search space is reduced), we see that for a final model compute cost of $C$ and width $N$ this procedure only introduces an additional factor of $O(C \log(N))$ to the computational requirements of hyperparameter tuning, with an optimum which is guaranteed to remain close to the true minimum at the largest model size. Hence, the final training run still consumes a significant fraction of the overall budget and the total tuning overhead remains modest. Ultimately, this telescoping scheme extends the standard muP "tune-up" procedure by ensuring that errors are governed by the largest calibration width $N_c$ (which can be chosen to be $O(N)$), as opposed to the smallest width $N_0$, while incurring only an overhead of $O(C \log(N))$.

### 3.4 Experiments

We validate our telescoping algorithm on a family of transformer models aimed at a final size of approximately 3.7B parameters. All models use a sequence length of 8192, a batch size of $2^{23} \approx 8$ million tokens and a depth of 34 layers. The smallest model in the family has head dimension 12, 1 key/value head, 2 query heads, a base embedding dimension of 256 and a base MLP dimension of 1024. We do not scale the AdamW $\epsilon$ parameter, following Yang *et al.* (2022) and Everett *et al.* (2024), as our models are sufficiently small. Additionally, although Yang (2019) demonstrates approximate hyperparameter transfer across model dimensions besides width, we note that related theoretical work (Large *et al.* (2024)) demonstrates that precise transfer along these dimensions requires architectural changes. Hence, we fix all other "scale" dimensions — depth, sequence length, batch size and training steps — and only vary width.

For the largest model (about 3.7B parameters), we train for $20,000$ steps, corresponding to 160B tokens—approximately 2.2× the Chinchilla-optimal budget (72B tokens) (Hoffmann *et al.* (2022)). Though this exceeds the nominal optimal budget, over-training can be beneficial when seeking a smaller inference footprint, often with negligible impact on final loss (McLeish *et al.* (2025)). The full hyperparameter sets are the point-wise products of $\mathcal{H}_q = \{2, 4, 8, 16, 20\}$, $\mathcal{H}_{kv} = \{1, 2, 4, 8, 10\}$, $\mathcal{D}_{\text{emb}} = \{256, 512, 1024, 2048, 2560\}$, $\mathcal{D}_{\text{mlp}} = \{1024, 2048, 4096, 8192, 10240\}$, $\mathcal{S} = \{0.07\text{B}, 0.20\text{B}, 0.67\text{B}, 2.40\text{B}, 3.67\text{B}\}$ with $\mathcal{H}_q$ the number of query heads, $\mathcal{H}_{kv}$ the number of kv heads, $\mathcal{D}_{\text{emb}}$ the embedding dimension, $\mathcal{D}_{\text{mlp}}$ the MLP dimension and $\mathcal{S}$ the model size. We fix head dimension at 128 for all experiments. All models are trained on a 50/50 mix of (i) high-quality web data derived from DCLM (Li *et al.* (2024)) and (ii) Python code derived from Stackv2 (Lozhkov *et al.* (2024)).

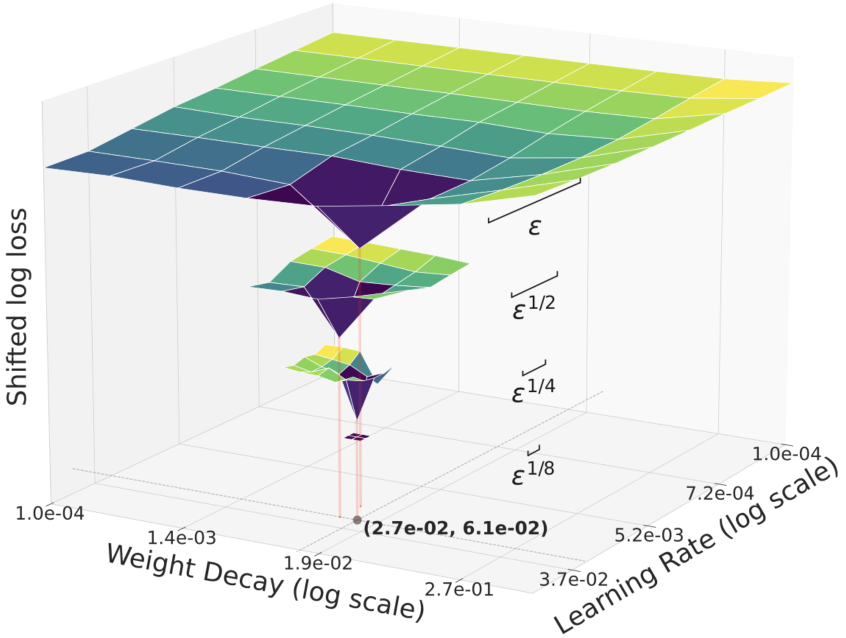

Figure 6: Telescoping algorithm applied to weight decay and learning rate. The loss for each model is shifted to a small offset and then presented on a logarithmic scale to exaggerate the minimum for visualization. The red vertical lines highlight the optimal hyperparameters for comparison. The narrowest model (smallest $n$) has the most grid points (the topmost surface) and the coarsest mesh with cell size $\epsilon = 1/(m-1)$ (see Eq. 5), with $m$ being the number of sampled points for each hyperparameter. Submeshes at each subsequent doubling of width (lower surfaces) shrink in size, highlighting how the minima of each surface remain close across widths. Yellow regions indicate higher losses; color scales differ per stage for clarity. The shift in the red lines, marking the minima at each of the four layers, is proportional to $\alpha/n$ in Eq. 4. The largest grid is $8 \times 8$, followed by $5 \times 5$, $3 \times 3$ and $1 \times 1$, approximately following our geometric schedule in Alg. 1. The third level, with cell size $\epsilon^{1/4}$ has extra points plotted to demonstrate that the true minimum is contained in the internal $3 \times 3$ subgrid. The final layer of the telescope, before training the final model, with cell size $\epsilon^{1/8}$ is $3 \times 3$ rather than $1 \times 1$ to demonstrate that the selected point is a true minimum (is locally flat).

We apply the telescoping algorithm (Alg. 1) from Sec. 3.3 to successively refine our hyperparameter sweeps while doubling the model width. At each stage, we reduce the search space geometrically, so that in the final sweep on the 2.40B model, the mesh points are spaced logarithmically with multiplicative steps of approximately 1.17 and 1.13 (comparable to standard mesh sizes, see e.g. the Appendix of Everett *et al.* (2024)). The variation in model performance at the two largest scales is not statistically significant, indicating that the re-meshing could have been stopped earlier without loss of accuracy (i.e. $N_c = 16$ in Alg. 1, but could have been chosen smaller, e.g. $N_c = 8$). Figure 13 summarizes the primary results and Figure 14 in the Appendix depicts the variance in loss per telescoping level, together with a standard power law fit to the minimum loss at each level. For clarity, the variance of the first grid is truncated in the figure.

We observe that each iteration of the telescoping procedure reduces the variance further; combined with the analysis in Sec. 3.3, this supports both precision (from the empirical variance in Eq. 5) and accuracy (from theoretical guarantees in Eq. 4). The final full model training on the 3.7B models achieves a final training loss of 1.61 nats when trained at the selected weight decay of $\lambda = 0.027$ (see Eq. 1) and muP base learning rate of $\eta_0 = 0.0612$ (see Sec. 3.1), outperforming all models at previous widths. We observe excellent agreement with a shifted power law ($R^2 \approx 1$), where the exponent for the parameter dependence is 0.31, close to the Chinchilla exponent of 0.34 (see Fig. 14).

# 4 Related Work

**Dethroning AdamW.** AdamW (Kingma and Ba, 2014; Loshchilov and Hutter, 2017) has long been the de facto king of the optimizers for large-scale neural network training. Recent research has shown promising results with new optimizers, especially with second-order methods based on non-diagonal preconditioners (Gupta *et al.*, 2018; Vyas *et al.*, 2024; Jordan *et al.*, 2024; Liu *et al.*, 2025), but also with first-order methods (Chen *et al.*, 2023; Shazeer and Stern, 2018; Zhai *et al.*, 2022; Zhao *et al.*, 2025). However, these works compare optimizers at fixed compute resources and do not conclusively show that an optimizer is better than AdamW in trading off compute and time resources. We address this limitation by explicitly showing that Muon expands AdamW's compute-time Pareto frontier. The notion of the compute-time Pareto frontier is first proposed by McCandlish *et al.* (2018) for studying batch size dynamics, but to our knowledge has not been used to compare optimizers.

**Impact of the batch size.** Most research on the impact of batch size on training relies on the idea of critical batch size, broadly construed as the largest batch size that allows for almost linear scaling beyond which we receive diminishing returns (Balles *et al.*, 2016; Goyal *et al.*, 2017; McCandlish *et al.*, 2018; Zhang *et al.*, 2019; Shallue *et al.*, 2019; Zhang *et al.*, 2024). While useful, such a point estimate is sensitive to noise and fails to describe the impact of the batch size in the post-critical regime, which can play a critical role in the compute-time tradeoff. We present a novel way to continuously measure the batch size advantage of an optimizer in the post-critical regime by monitoring the ratio of token consumptions.

**Practical aspects of muP.** There are many existing works that consider muP (Ishikawa and Karakida (2023); Blake *et al.* (2024); Haas *et al.* (2024); Yaida (2022); Dinan *et al.* (2023); Everett *et al.* (2024); Yang *et al.* (2022); Dey *et al.* (2024); Halverson (2024); Meta AI (2024)), and so it is worth asking in what ways our study differs. Generally speaking, muP has been shown to be theoretically correct, the key insight being that it is possible to relate the infinite-width limit to finite-width network behavior. It has also been empirically demonstrated to work, although existing studies have omitted rigorous investigations and analyses of the errors that we have discussed in this paper. While the primary intent of Everett *et al.* (2024) was to address a particular assumption made in the derivation of muP, another major contribution of the study was the large number of experiments they ran which demonstrate the behavior of muP in practice. Yaida (2022) discusses different scaling parameterizations and in Dinan *et al.* (2023) they discuss their empirical performance.

Meta AI (2024) highlights a new technique, "MetaP", which while not described, emphasizes the need for precise hyperparameter transfer across other scale dimensions (i.e. depth, batch size, sequence length and training steps), which are shown to hold only approximately in Yang *et al.* (2022). Finally, Ishikawa and Karakida (2023) proposed techniques for applying muP to second order optimization (i.e. Shampoo and KFAC), however experimental demonstrations of muP together with Muon have yet to have been empirically validated Moonshot (2025). Specifically, despite Muon's close relationship to Shampoo (see. App. B), we have shown in Sec. 3.4 that the same muP scaling that is used for AdamW in Yang *et al.* (2022) (see Table 5) works for Muon. Due to the considerable similarities between Muon's hyperparameter transfer properties and muP's hyperparameter transfer properties discussed in Bernstein (2025) our observation of Muon's simple compatibility with muP is consistent with existing literature.

# 5 Conclusion

This paper has addressed two practical questions that arise in language model pretraining: (1) which optimizer delivers the best tradeoff between compute and time resources, and (2) how to tune that optimizer without expending prohibitive compute. For the first question, we have shown that Muon expands AdamW's Pareto frontier on the compute-time plane, enlarging the practitioner's flexibility in resource allocation. We have given an account for this advantage by analyzing the token ratio (2), which measures the relative data efficiency of Muon at large batch sizes. Concretely, Muon requires 10–15 % fewer tokens than AdamW to reach an identical loss and converts these savings into faster wall-clock convergence, with the advantage staying constant or growing as the batch size increases. The findings are validated by extensive experiments across five model sizes (100M–4B parameters), two data modalities, and an order of several decades of variation in global batch size. These results establish Muon as a drop-in successor to AdamW for second-order optimization at scale.

For the second question, we have given the first affirmative answer to the open question posed by Moonshot (2025) and Liu *et al.* (2025): does the maximal-update parametrization (muP) remain valid when used together with Muon? Our experiments confirm that muP transfers hyperparameters cleanly up to 3.7B-parameter models at sequence length $8192$, for both learning rate and coupled weight decay. Building on this observation, we have introduced a telescoping hyperparameter sweep that contracts the search grid at each width doubling, bounding tuning overhead by a factor of $O(C \log N)$ where $N$ is the final model width and $C$ is the cost of training the full model, while quantifying all known sources of transfer error (Eq. 4, Eq. 5, Fig. 13). In practice, more than 20% of the total compute budget is now devoted to the final, full-scale training run while guaranteeing near-optimal hyperparameters.

Taken together, these contributions give a unified recipe: Muon optimization, muP scaling, and telescoping hyperparameter transfer. The recipe delivers strictly superior data efficiency, shorter training time, and near-negligible tuning cost compared with the AdamW baseline. The evidence presented here promotes Muon from a promising alternative to a robust, drop-in replacement for AdamW, and converts second-order optimization—long regarded as computationally extravagant—into a practical default for industry-scale language model pretraining.

## References

Anil, R., Gupta, V., Koren, T., Regan, K., and Singer, Y. (2020). Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*.

Balles, L., Romero, J., and Hennig, P. (2016). Coupling adaptive batch sizes with learning rates. *arXiv preprint arXiv:1612.05086*.

Bernstein, J. (2025). Deriving muon.

Bernstein, J. and Newhouse, L. (2024). Old optimizer, new norm: An anthology. *arXiv preprint arXiv:2409.20325*.

Björck, Å. and Bowie, C. (1971). An iterative algorithm for computing the best estimate of an orthogonal matrix. *SIAM Journal on Numerical Analysis*, **8**(2), 358–364.

Blake, C., Eichenberg, C., Dean, J., Balles, L., Prince, L. Y., Deiseroth, B., Cruz-Salinas, A. F., Luschi, C., Weinbach, S., and Orr, D. (2024). u-$\mu$p: The unit-scaled maximal update parametrization. *arXiv preprint arXiv:2407.17465*.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., *et al.* (2023). Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, **36**, 49205–49233.

Dey, N., Bergsma, S., and Hestness, J. (2024). Sparse maximal update parameterization: A holistic approach to sparse training dynamics. *arXiv preprint arXiv:2405.15743*.

Dinan, E., Yaida, S., and Zhang, S. (2023). Effective theory of transformers at initialization. *arXiv preprint arXiv:2304.02034*.

Everett, K., Xiao, L., Wortsman, M., Alemi, A. A., Novak, R., Liu, P. J., Gur, I., Sohl-Dickstein, J., Kaelbling, L. P., Lee, J., *et al.* (2024). Scaling exponents across parameterizations and optimizers. *arXiv preprint arXiv:2407.05872*.

Fetterman, A. J., Kitanidis, E., Albrecht, J., Polizzi, Z., Fogelman, B., Knutins, M., Wróblewski, B., Simon, J. B., and Qiu, K. (2023). Tune as you scale: Hyperparameter optimization for compute efficient training. *arXiv preprint arXiv:2306.08055*.

Gemma Team (2025). Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Gupta, V., Koren, T., and Singer, Y. (2018). Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR.

Haas, M., Xu, J., Cevher, V., and Vankadara, L. C. (2024). Effective sharpness aware minimization requires layerwise perturbation scaling. In *High-dimensional Learning Dynamics 2024: The Emergence of Structure and Reasoning*.

Halverson, J. (2024). Tasi lectures on physics for machine learning. *arXiv preprint arXiv:2408.00082*.

Hanin, B. (2023). Random neural networks in the infinite width limit as gaussian processes. *The Annals of Applied Probability*, **33**(6A), 4798–4819.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., *et al.* (2022). Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Ishikawa, S. and Karakida, R. (2023). On the parameterization of second-order optimization effective towards the infinite width. *arXiv preprint arXiv:2312.12226*.

Jordan, K., Jin, Y., Boza, V., Jiacheng, Y., Cesista, F., Newhouse, L., and Bernstein, J. (2024). Muon: An optimizer for hidden layers in neural networks.

Kennedy, M. C. and O'Hagan, A. (2000). Predicting the output from a complex computer code when fast approximations are available. *Biometrika*, **87**(1), 1–13.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kovarik, Z. (1970). Some iterative methods for improving orthonormality. *SIAM Journal on Numerical Analysis*, **7**(3), 386–389.

Large, T., Liu, Y., Huh, M., Bahng, H., Isola, P., and Bernstein, J. (2024). Scalable optimization in the modular norm. *arXiv preprint arXiv:2405.14813*.

Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S. Y., Bansal, H., Guha, E., Keh, S. S., Arora, K., *et al.* (2024). Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, **37**, 14200–14282.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, **18**(185), 1–52.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., *et al.* (2024). Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Liu, J., Su, J., Yao, X., Jiang, Z., Lai, G., Du, Y., Qin, Y., Xu, W., Lu, E., Yan, J., *et al.* (2025). Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., *et al.* (2024). Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. (2018). An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*.

McLeish, S., Kirchenbauer, J., Miller, D. Y., Singh, S., Bhatele, A., Goldblum, M., Panda, A., and Goldstein, T. (2025). Gemstones: A model suite for multi-faceted scaling laws. *arXiv preprint arXiv:2502.06857*.

Meta AI (2024). Llama 4: Advancing multimodal intelligence. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`. Accessed: 2025-04-10.

MLCommons (2025). Algoperf: Training algorithms benchmark results. `https://mlcommons.org/benchmarks/algorithms/`. Accessed: 2025-04-19.

Moonshot, K. (2025). Llama 4 + meta ai tweet. `https://x.com/Kimi_Moonshot/status/1897929976948965870`. Tweet, accessed: 2025-04-10.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. (2019). Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, **20**(112), 1–49.

Shazeer, N. and Stern, M. (2018). Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

Vyas, N., Morwani, D., Zhao, R., Kwun, M., Shapira, I., Brandfonbrener, D., Janson, L., and Kakade, S. (2024). Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*.

Wang, X. and Aitchison, L. (2024). How to set adamw's weight decay as you scale model and dataset size. *arXiv preprint arXiv:2405.13698*.

Yaida, S. (2022). Meta-principled family of hyperparameter scaling strategies. *arXiv preprint arXiv:2210.04909*.

Yang, G. (2019). Scaling limits of wide neural networks with weight sharing: Gaussian process behavior, gradient independence, and neural tangent kernel derivation. *arXiv preprint arXiv:1902.04760*.

Yang, G., Hu, E. J., Babuschkin, I., Sidor, S., Liu, X., Farhi, D., Ryder, N., Pachocki, J., Chen, W., and Gao, J. (2022). Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer. *arXiv preprint arXiv:2203.03466*.

Yang, G., Simon, J. B., and Bernstein, J. (2023). A spectral condition for feature learning. *arXiv preprint arXiv:2310.17813*.

Zhai, X., Kolesnikov, A., Houlsby, N., and Beyer, L. (2022). Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12104–12113.

Zhang, G., Li, L., Nado, Z., Martens, J., Sachdeva, S., Dahl, G., Shallue, C., and Grosse, R. B. (2019). Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. *Advances in neural information processing systems*, **32**.

Zhang, H., Morwani, D., Vyas, N., Wu, J., Zou, D., Ghai, U., Foster, D., and Kakade, S. (2024). How does critical batch size scale in pre-training? *arXiv preprint arXiv:2410.21676*.

Zhao, R., Morwani, D., Brandfonbrener, D., Vyas, N., and Kakade, S. M. (2025). Deconstructing what makes a good optimizer for autoregressive language models. In *The Thirteenth International Conference on Learning Representations*.

# A Contributions

### Core Contributors

Ishaan Shah[⋆]
Anthony M. Polloreno[⋆]
Karl Stratos[⋆]
Philip Monk
Ashish Vaswani

### Contributors

Adarsh Chaluvaraju
Andrew Hojel
Andrew Ma
Anil Thomas
Ashish Tanwer
Darsh J Shah
Khoi Nguyen
Kurt Smith
Michael Callahan
Michael Pust
Mohit Parmar
Peter Rushton
Platon Mazarakis
Ritvik Kapila
Saurabh Srivastava
Somanshu Singla
Tim Romanski
Yash Vanjani

[⋆] Equal contribution

# B Reduction of Shampoo and Soap to Muon

It is well known that Shampoo (Gupta *et al.*, 2018) without momentum is equivalent to Muon (Jordan *et al.*, 2024). At its core, Shampoo uses the following gradient transformation

$$O_t = \mathbf{E}\left[GG^\top\right]^{-1/4} G_t \mathbf{E}\left[G^\top G\right]^{-1/4}$$

where $G_t \in \mathbb{R}^{m \times n}$ is the gradient on the current batch and $G \in \mathbb{R}^{m \times n}$ is the gradient on a random batch. The second moments (assumed to be full rank) can be estimated in an online fashion using bias-corrected exponential moving average (EMA). If we instead consider a point estimate $\mathbf{E}\left[GG^\top\right] \approx G_t G_t^\top$ and $\mathbf{E}\left[G^\top G\right] \approx G_t^\top G_t$, we have

$$O_t \approx (G_t G_t^\top)^{-1/4} G_t (G_t^\top G_t)^{-1/4} = U\Sigma^{-1/2}U^\top U\Sigma V^\top V\Sigma^{-1/2}V^\top = UV^\top$$

where $G_t = U\Sigma V^\top$ denotes an SVD. A less well-known connection is between Soap (Vyas *et al.*, 2024) and Muon. Soap performs Adam in a second-order basis. Specifically, let $\mathbf{E}\left[GG^\top\right] = Q_L \Lambda_L Q_L^\top$ and $\mathbf{E}\left[GG^\top\right] = Q_R \Lambda_R Q_R^\top$ denote the eigendecomposition of the second moments. Soap uses the following gradient transformation

$$\begin{aligned}
\bar{G}_t &= Q_L^\top G_t Q_R \\
M_t &= \mathrm{UpdateEMA}\left(\bar{G}_t\right) \\
V_t &= \mathrm{UpdateEMA}\left(\bar{G}_t^{\langle 2 \rangle}\right) \\
\bar{O}_t &= \frac{M_t}{\sqrt{V_t}} \\
O_t &= Q_L \bar{O}_t Q_R^\top
\end{aligned}$$

where $A^{\langle 2 \rangle}$ is the elementwise square of $A$. Consider again computing $Q_L, Q_R$ from a point estimate. Then $Q_L = U$ and $Q_R = V$ since $G_t G_t^\top = U\Sigma^2 U^\top$ and $G_t^\top G_t = V\Sigma^2 V^\top$. Further computing $M_t, V_t$ from a point estimate, we have (assuming $m \leq n$)

$$\begin{aligned}
\bar{G}_t &= \Sigma \\
M_t &= \Sigma \\
V_t &= \Sigma^2 \\
\bar{O}_t &= I_{m \times m} \\
O_t &= UV^\top
\end{aligned}$$

Thus Shampoo and Soap can be reduced to Muon under these simplifying assumptions. In particular, the motivation for Shampoo as an implicit approximation of the inverse square root of the empirical Fisher matrix (which itself can be viewed as an approximation of the inverse Hessian) transfers to Muon.

# C Initial Verification of Muon's Performance

We find that Muon consistently achieves a lower training loss and faster convergence across different model sizes and datasets. We give a concrete example with 1B model size on DCLM in Figure 7. We take the best training runs (i.e., the hyperparameter configurations resulting in the lowest final loss) under Adam and Muon and plot the training loss as a function of steps and hours (smoothed using EMA with coefficient 0.95). It is clear that Muon strictly lower bounds AdamW on these curves, even to the end of training far beyond the Chinchilla-optimal budget (i.e., there is no crossover). It is notable that the wall time advantage holds with our minimal Muon implementation and may further improve with a more sophisticated version (e.g., using lower precision for the first moment). Our finding strengthens and expands the existing evidence on Muon's promising utility for pretraining (Jordan *et al.*, 2024; Liu *et al.*, 2025).

# D Deeper Analysis of the Critical Batch Size

Increasing the batch size is one of the easiest ways to accelerate training on data parallel hardware. A great deal of research focuses on the idea of "critical" batch size, which is broadly defined as the

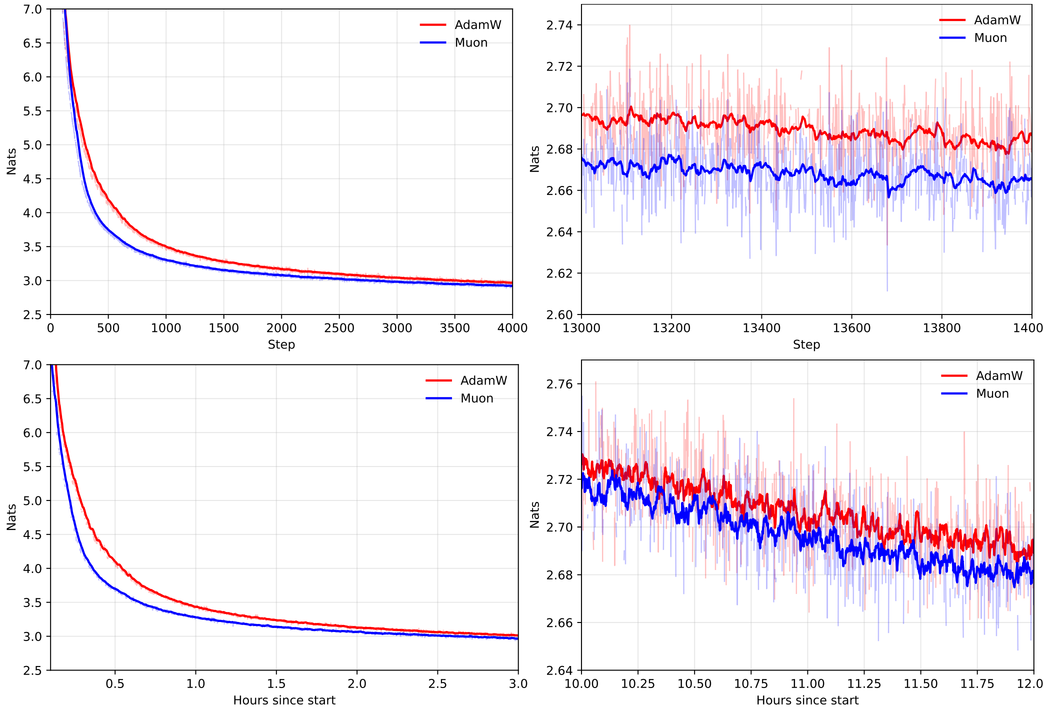

Figure 7: Comparison of the best 1B model training runs on DCLM under AdamW and Muon in steps and wall time. (Top) Muon has a consistently lower training loss compared to Adam given the same number of steps. The difference does not vanish at the end of training (0.02 nats at 2.5× Chinchilla-optimal). (Bottom) Muon obtains a target loss faster than Adam.

largest batch size that allows for almost linear or "perfect" scaling (e.g., doubling the batch size halves the number of steps) beyond which we receive diminishing returns.

Despite the simple intuition, formalizing the critical batch size can be nontrivial (McCandlish *et al.*, 2018). A more direct approach is to measure the number of steps $S_L(B)$ required to reach a target loss $L$ as a function of the batch size $B$. Let $T_L(B) = B \times S_L(B)$ denote the corresponding number of tokens consumed. We can define the critical batch size as a value $B_L^\star$ such that (1) $T_L(B) = T_L^\star$ remains constant for $B \leq B_L^\star$ where $T_L^\star$ represents the minimum number of tokens required to reach loss $L$, and (2) $T_L(B) = \mathrm{inc}_L(B)$ is some increasing function for $B > B_L^\star$.[⋆] In the perfect scaling regime, we have $S_L(B) = T_L^\star B^{-1}$ which is linearly decreasing with slope $-1$ in log-log scale. Researchers often gauge this "phase transition" (i.e., kink) from an empirical plot, and argue that an optimizer enjoys a larger critical batch size if the transition seems to happen at a larger batch size (Zhang *et al.*, 2019; Vyas *et al.*, 2024).

Figure 8 shows the step curves for AdamW and Muon (in log-log scale). While the Muon curve lower bounds the AdamW curve in general, they are limiting for the purpose of critical batch size study for the following reasons. First, they do not always exhibit a clear phase transition (e.g., there may be no major kink, or there may be multiple), particularly at a low target loss. Second, the semantics of the plot is tightly coupled with the slope. If the slope is larger than $-1$, it implies that we should use a smaller batch size regardless of any kink since we are already in the regime of diminishing returns. Conversely, if the slope is smaller than $-1$, there can be several kinks but none of them is a critical batch size. Thus without reporting the actual slope value, the visual presence of phrase transitions alone can be misleading.

[⋆] The specific form of $\mathrm{inc}_L(B)$ in the post-critical regime is not prescribed (other than generally increasing). Appendix F gives a simple parametric model for completeness.

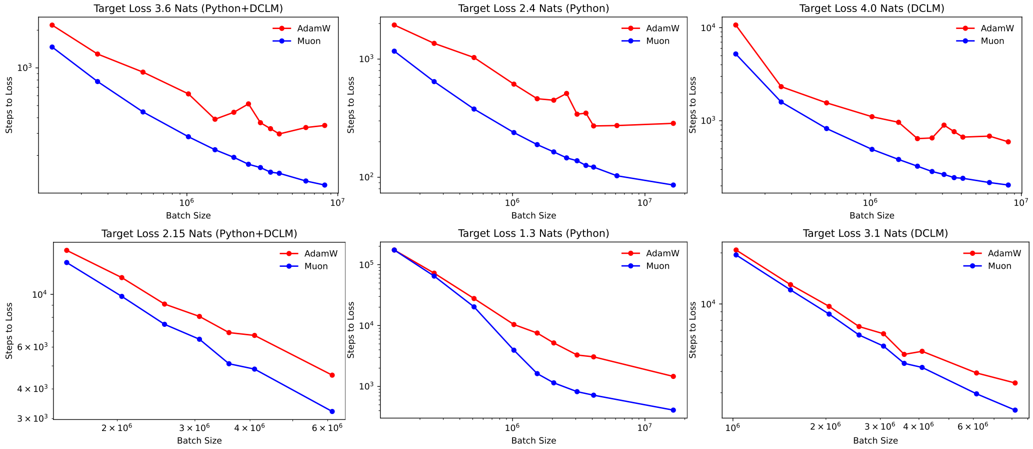

Figure 8: Steps-to-loss vs batch size plots with 500M parameter models for target losses near convergence (log-log scale).

### D.1 Token-optimal batch size

To facilitate a more effective measurement of optimal batch size, we can examine the tokens-to-loss $T_L(B)$ rather than steps-to-loss $S_L(B)$. By definition, the tokens-to-loss curve is flat in the perfect scaling regime and increasing in the post-critical regime (Figure 4 left). We may then consider a "token-optimal" batch size as

$$B_{\text{tok},L} = \max\left\{B:\ T_L(B) = \min_{B'} T_L(B')\right\} \tag{6}$$

(i.e., the largest data-optimal batch size). Unlike the kinks in the steps-to-loss plot, $B_{\text{tok},L}$ is always uniquely defined and allows for convenient measurement. In an idealized setting where the critical batch size assumption holds, $B_{\text{tok},L}$ coincides with $B^\star_L$.

We can monitor the evolution of (6) throughout training by plotting it across various loss thresholds near convergence. Figure 9, 10, and 11 shows such plots for AdamW and Muon ablating the architecture, dataset, and model size. We see that the token-optimal batch size increases during training for both optimizers, which is consistent with the predicted behavior of critical batch size (McCandlish *et al.*, 2018). Muon generally achieves a larger token-optimal batch sizes across these ablations.

While useful, the token-optimal batch size is still insufficient for our purpose of fully characterizing the batch size behavior of an optimizer. First, it is brittle under noise. We can easily construct two nearly identical tokens-to-loss curves where one has a much larger token-optimal batch size, but the advantage flips with an infinitesimal perturbation. Second, much more seriously, it does not capture the batch size benefit in the over-large batch size regime. For instance, on DCLM (Figure 10 bottom row), both AdamW and Muon have the same token-optimal batch size of 3.5M at loss 3.25. However, the token consumption increases much more slowly for Muon compared to AdamW as we further push the batch size beyond 3.5M. We capture this large-batch data efficiency behavior with the token ratio (2).

## E Additional Compute-Time Tradeoff Plots

Figure 12 overlays the plots for different thresholds. The curve moves upward as the target loss decreases, corresponding to the fact that we need more time and compute resources to achieve lower losses.

## F A Simple Parametric Model of the Critical Batch Size

We assume that the relationship between the number of steps $S_L(B)$ and the batch size $B$ is piecewise linear in log-log scale. This assumption is strong but may be motivated by empirical observations

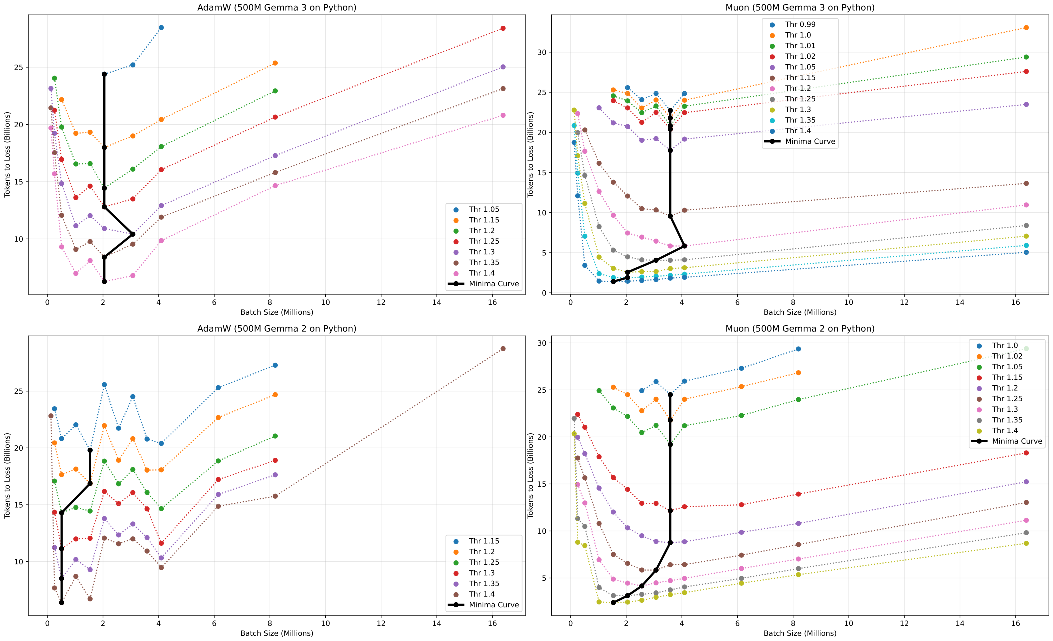

Figure 9: Token-optimal batch size plots: architecture ablation (Gemma 3 vs Gemma 2)

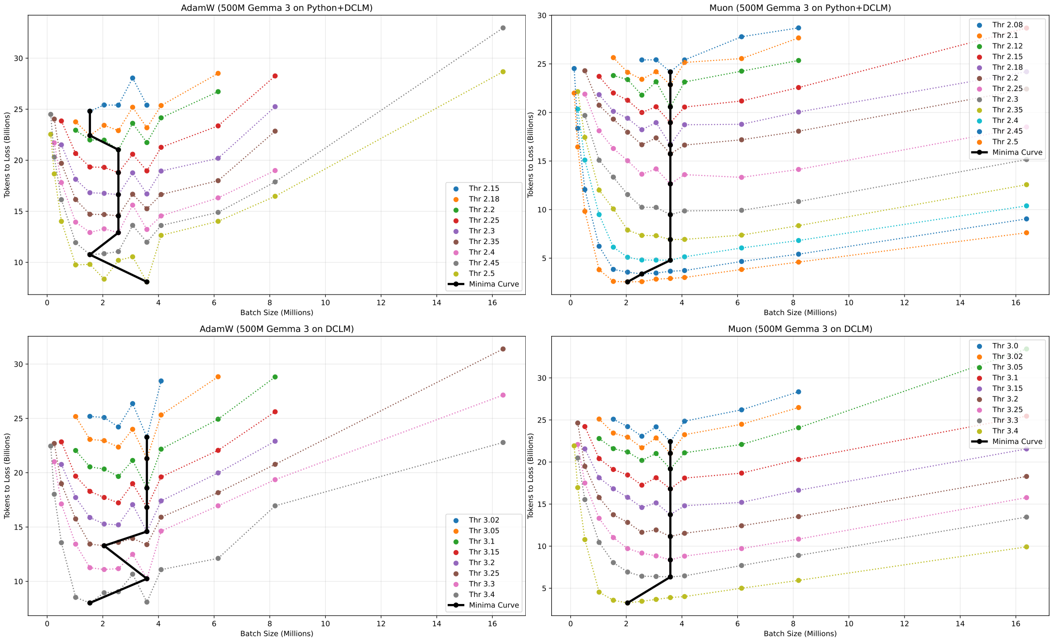

Figure 10: Token-optimal batch size plots: dataset ablation (Python+DCLM and DCLM)

in existing works which insinuate this behavior (Zhang *et al.*, 2019; Vyas *et al.*, 2024). Under this assumption, it follows that

$$\log S_L(B) = \begin{cases} -\log B + b_1 & \text{for } B \leq B_L^\star \\ m \log B + b_2 & \text{for } B > B_L^\star, \text{ where } -1 < m < 0 \end{cases}$$

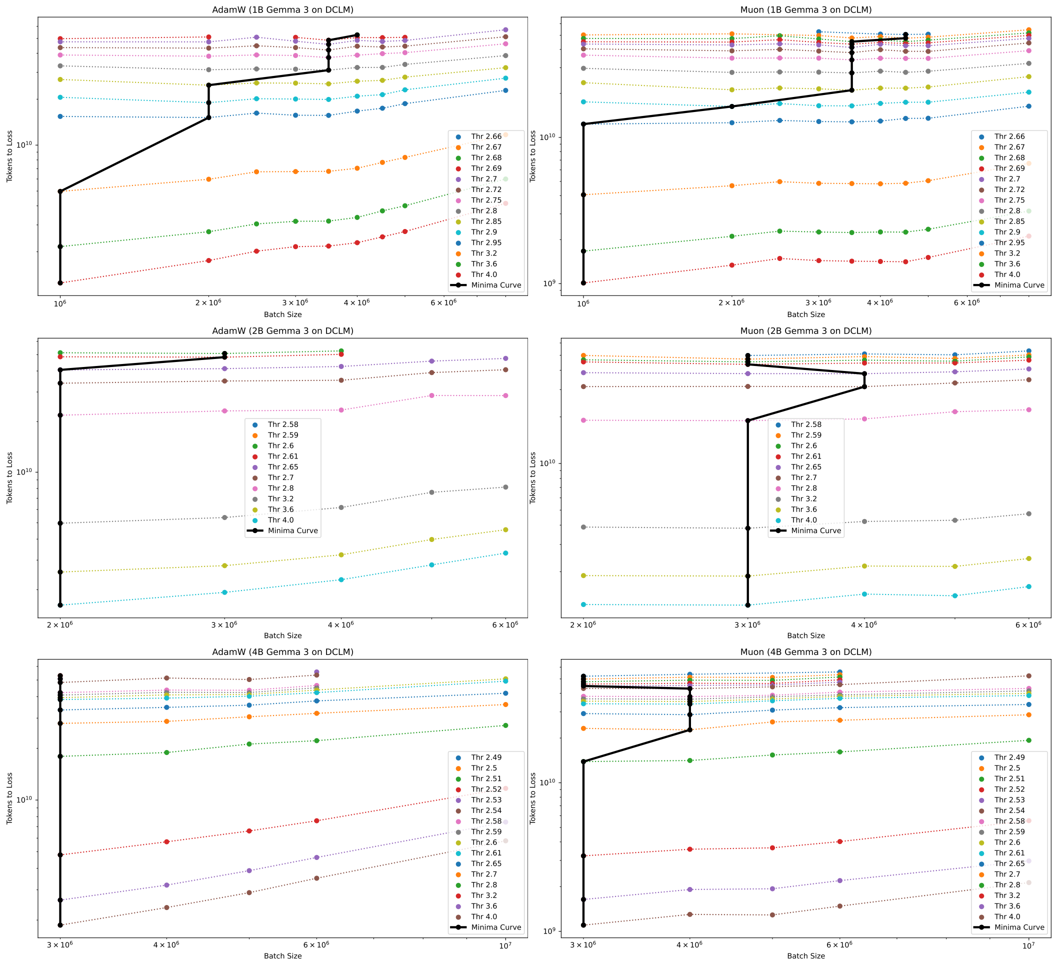

Figure 11: Token-optimal batch size plots: model size ablation (1B, 2B, and 4B)

with some intercepts $b_1, b_2$ such that $\log S_L(B)$ remains continuous. From $S_L(B) = T_L^\star B^{-1}$ we know $b_1 = \log T_L^\star$. We then have the following parametric forms for the number of steps and tokens:

$$S_L(B) = \begin{cases} e^{b_1} B^{-1} & \text{for } B \le B_L^\star \\ e^{b_2} B^m & \text{for } B > B_L^\star \end{cases} \qquad T_L(B) = \begin{cases} e^{b_1} & \text{for } B \le B_L^\star \\ e^{b_2} B^{m+1} & \text{for } B > B_L^\star \end{cases}$$

Let the subscripts $A, M$ denote AdamW and Muon. Assuming $B_{L,A}^\star < B_{L,M}^\star$, the token ratio $R_L(B) = T_{L,A}(B)/T_{L,M}(B)$ has the following form:

$$R_L(B) = \begin{cases} e^{b_{1,A} - b_{1,M}} & \text{for } B \le B_{L,A}^\star \\ e^{b_{2,A} - b_{1,M}} B^{m_A + 1} & \text{for } B_{L,A}^\star < B \le B_{L,M}^\star \\ e^{b_{2,A} - b_{2,M}} B^{m_A - m_B} & \text{for } B > B_{L,M}^\star \end{cases}$$

In particular, $R_L(B)$ does not vanish in the post-critical regime iff $m_A \ge m_B$.

# G Muon Implementation

We adapted the experimental Muon implementation on Optax.[⋆] Specifically, we modified the application of Newton-Schulz to handle the transformer weights shaped differently for different layers under `jax.lax.scan`. We maintain the same implementation and the default hyperparameters for Newton-Schulz given by Jordan *et al.* (2024).

[⋆] `https://github.com/google-deepmind/optax/blob/6bd761c00d839ca363bffac2584888ac797fc69d/optax/contrib/_muon.py`.

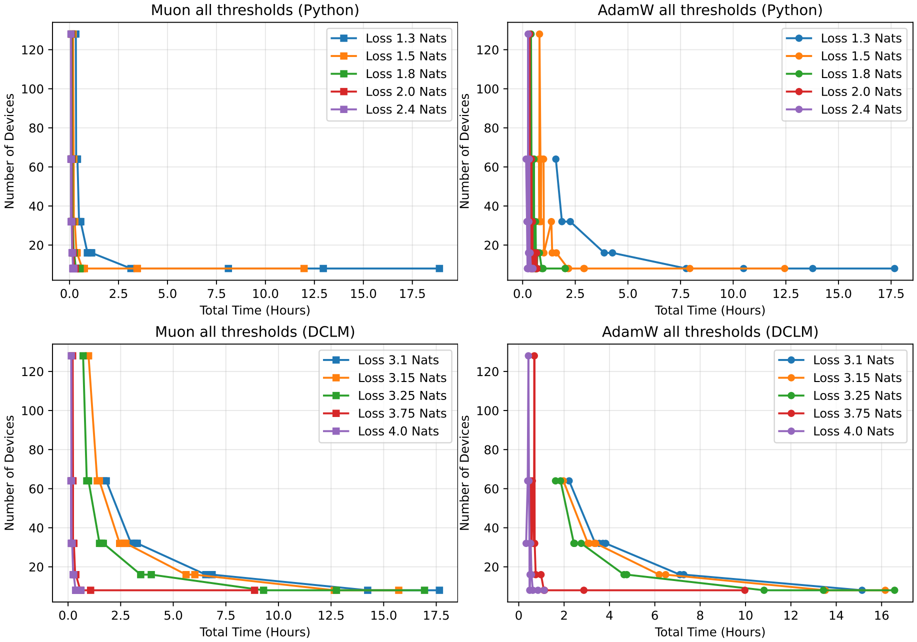

Figure 12: Compute-time tradeoff plots for all thresholds between Muon vs AdamW: Python (top) and DCLM (bottom).

```
from typing import NamedTuple, Optional, Union

import chex
import jax
import jax.numpy as jnp
import optax

from optax import tree_utils as otu
from optax._src import base
from optax._src import combine
from optax._src import transform
from optax._src import utils

from collections.abc import Callable
from typing import Any, Optional, Union, Tuple
MaskOrFn = Optional[Union[Any, Callable[[base.Params], Any]]]


def orthogonalize_matrix(
      x: jax.Array,  # (batch_size, d, D)
      ns_steps: int = 5,
      eps: float = 1e-7,
) -> jax.Array:
  a, b, c = (3.4445, -4.7750, 2.0315)
  transposed = False
  if x.shape[1] > x.shape[2]:
    x = x.transpose(0, 2, 1)
    transposed = True

  def newton_schulz_iterator(X: jax.Array) -> jax.Array:
    A = X @ X.transpose(0, 2, 1)
    B = b * A + c * A @ A
    return a * X + B @ X

  x /= jnp.linalg.norm(x, axis=(1, 2), keepdims=True) + eps
  x = jax.lax.fori_loop(0, ns_steps, lambda _, x: newton_schulz_iterator(x), x)
  if transposed:
    x = x.transpose(0, 2, 1)
  return x


def orthogonalize_via_newton_schulz_scan_attn_out(
    x: jax.Array,  # (num_heads, num_layers, dim_head, dim)
    ns_steps: int = 5,
    eps: float = 1e-7,
) -> jax.Array:
  (num_heads, num_layers, dim_head, dim) = x.shape
  x = x.transpose(1, 0, 2, 3)  # (num_layers, num_heads, dim_head, dim)
  x = x.reshape(num_layers, num_heads * dim_head, dim)
  x = orthogonalize_matrix(x, ns_steps, eps)
  x = x.reshape(num_layers, num_heads, dim_head, dim)
  x = x.transpose(1, 0, 2, 3)  # (num_heads, num_layers, dim_head, dim)
  return x


def orthogonalize_via_newton_schulz_scan_attn_qkv(
    x: jax.Array,  # (dim, num_layers, num_heads, dim_head)
    ns_steps: int = 5,
    eps: float = 1e-7,
) -> jax.Array:
  (dim, num_layers, num_heads, dim_head) = x.shape
  x = x.transpose(1, 0, 2, 3)  # (num_layers, dim, num_heads, dim_head)
  x = x.reshape(num_layers, dim, num_heads * dim_head)
  x = orthogonalize_matrix(x, ns_steps, eps)
  x = x.reshape(num_layers, dim, num_heads, dim_head)
  x = x.transpose(1, 0, 2, 3)  # (dim, num_layers, num_heads, dim_head)
  return x


def orthogonalize_via_newton_schulz_scan_mlp(
    x: jax.Array,  # (dim, num_layers, dim_hidden)
    ns_steps: int = 5,
    eps: float = 1e-7,
) -> jax.Array:
  x = x.transpose(1, 0, 2)  # (num_layers, dim, dim_hidden)
  x = orthogonalize_matrix(x, ns_steps, eps)
  x = x.transpose(1, 0, 2)  # (dim, num_layers, dim_hidden)
  return x


def apply_orthogonalization_logic(x, path, ns_steps, eps, base_scale: float):
    path_str = "/".join(path).lower()

    # Name-based logic, may need to change for more robust handling.
    if "mlp" in path_str:
        assert x.ndim == 3
        (d1, _, d2) = x.shape
        x_orth = orthogonalize_via_newton_schulz_scan_mlp(x, ns_steps, eps)
    elif "attention" in path_str and "out" not in path_str:
        assert x.ndim == 4
        (d1, _, num_heads, dim_head) = x.shape
        d2 = num_heads * dim_head
        x_orth = orthogonalize_via_newton_schulz_scan_attn_qkv(x, ns_steps, eps)
    elif "attention" in path_str and "out" in path_str:
        assert x.ndim == 4
        (num_heads, _, dim_head, d1) = x.shape
        d2 = num_heads * dim_head
        x_orth = orthogonalize_via_newton_schulz_scan_attn_out(x, ns_steps, eps)
    else:
        raise ValueError(f"Unidentifiable path for Muon: {path_str}")

    scale = base_scale * jnp.sqrt(jnp.maximum(d1, d2)) if base_scale > 0.0 else 1.0
    return x_orth * scale


def orthogonalize_tree(
    pytree: Any,
    ns_steps,
    eps,
    base_scale,
    path: Tuple[str, ...] = (),
) -> Any:
    if isinstance(pytree, optax.MaskedNode):
        return pytree
    elif isinstance(pytree, dict):
        return {k: orthogonalize_tree(v, ns_steps, eps, base_scale, path + (k,)) for k, v in pytree.items()}
    else:
        return None if pytree is None else apply_orthogonalization_logic(pytree, path, ns_steps, eps, base_scale)


class MuonState(NamedTuple):
  mu: base.Updates


def scale_by_muon(
    momentum: float = 0.95,
    *,
    mu_dtype: Optional[chex.ArrayDType] = None,
    nesterov: bool = True,
    eps: float = 1e-7,
    ns_steps: int = 5,
    base_scale: float = 0.2,
) -> base.GradientTransformation:
  mu_dtype = utils.canonicalize_dtype(mu_dtype)

  def init_fn(params):
    mu = otu.tree_zeros_like(params, dtype=mu_dtype)  # First moment
    return MuonState(mu=mu)

  def update_fn(updates, state, params=None):
    del params

    def momentum_grad(grad, buf):
      return grad + momentum * buf if grad is not None else None

    new_mu = jax.tree.map(momentum_grad, updates, state.mu, is_leaf=lambda x: x is None)
    mu = jax.tree.map(momentum_grad, updates, new_mu, is_leaf=lambda x: x is None) if nesterov else new_mu
    mu_orth = orthogonalize_tree(mu, ns_steps, eps, base_scale)

    return mu_orth, MuonState(mu=otu.tree_cast(new_mu, mu_dtype))

  return base.GradientTransformation(init_fn, update_fn)


def my_param_labels(params):
    def recurse(subtree, path=()):
        if isinstance(subtree, dict):
            return {k: recurse(v, path + (k,))  for k, v in subtree.items()}
        else:
            if any((’norm’ in k or ’logits’ in k or ’embedding’ in k) for k in path):
                return ’adam’
            else:
                return ’muon’
    return recurse(params)


def muon(
    learning_rate: base.ScalarOrSchedule,
    momentum: float = 0.95,
    *,
    mu_dtype: Optional[chex.ArrayDType] = None,
    nesterov: bool = True,
    eps: float = 1e-7,
    ns_steps: int = 5,
    base_scale: float = 0.2,
    adam_b1: float = 0.95,
    adam_b2: float = 0.95,
    adam_eps: float = 1e-8,
    weight_decay: float = 0.1,
    mask: MaskOrFn = None,
) -> base.GradientTransformation:
  return combine.chain(
    combine.multi_transform(
      transforms={
        ’muon’: scale_by_muon(
          momentum=momentum,
          mu_dtype=mu_dtype,
          nesterov=nesterov,
          eps=eps,
          ns_steps=ns_steps,
          base_scale=base_scale,
        ),
        ’adam’: transform.scale_by_adam(
          b1=adam_b1,
          b2=adam_b2,
          eps=adam_eps,
          eps_root=0,
          mu_dtype=mu_dtype,
          nesterov=nesterov,
        )
      },
      param_labels=lambda params: my_param_labels(params),
    ),
    transform.add_decayed_weights(weight_decay, mask),
    transform.scale_by_learning_rate(learning_rate),
  )
```

# H Parameterization of Telescoping and Distribution of Losses

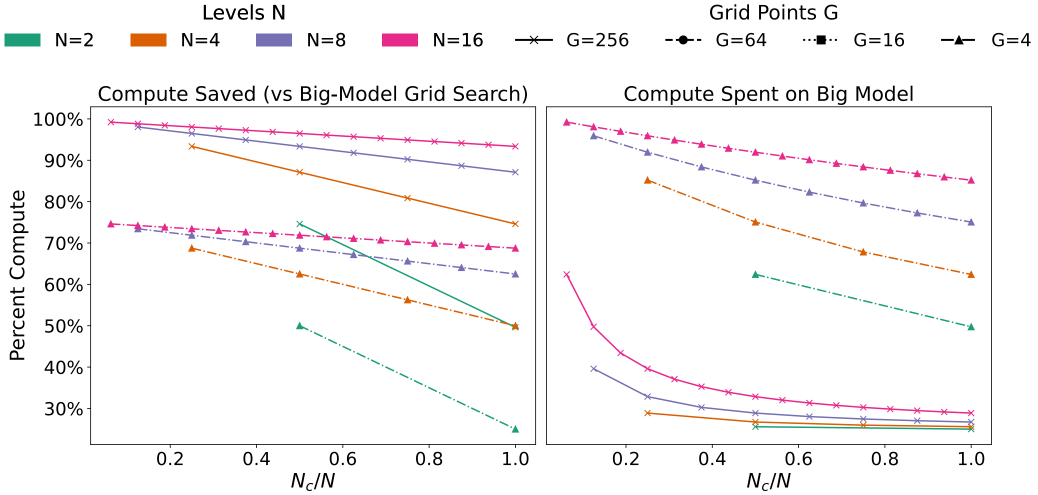

Figure 13: $N_c$ is the calibration level of the telescope - the level corresponding to the largest width we perform the telescoping up to. Two figures of merit: (left) the percentage of compute saved using telescoping muP as opposed to performing a full grid search on the full model size, for $G$ grid points, and $N$ levels the base model width. Savings are largest for large grid sweeps ($G$ large), large models ($N$ large), and small truncation ratios ($N_c/N$ small). (right) The percentage of the total compute spent on the final model run. Because we perform the same amount of compute at each stage of the telescope, this is maximized for small $N_c/N$, following the procedure suggested by Yang *et al.* (2022). This measures a form of exploration-exploitation trade off.

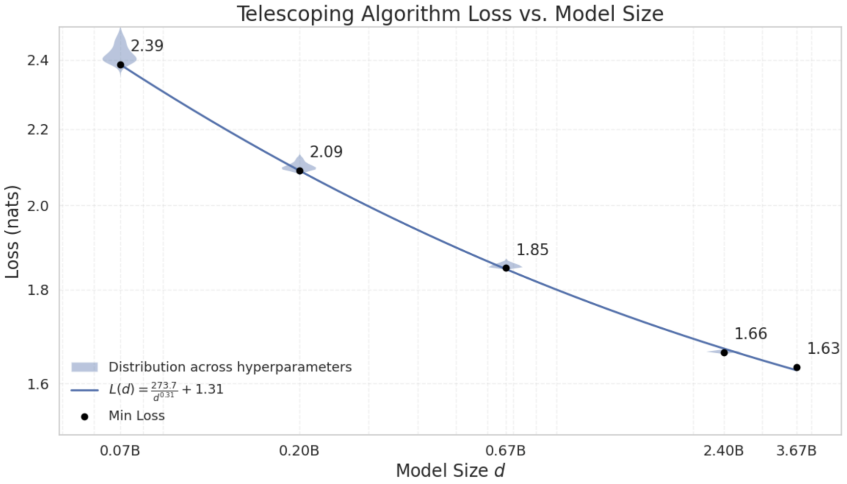

Figure 14: The distribution of loss values at each level of the telescope in Fig. 6, truncated at 2.5 to omit outliers from the initial broad sweep. We note that as the telescope level increases, the distribution becomes tighter around its mean, demonstrating that we are producing a precise estimate of the optimal hyperparameters, and the loss continues to decrease demonstrating that we maintain accuracy. The minimum loss values at each point fit well ($R^2 \approx 1$) to a Chinchilla style loss function ($L(d) = A/d^\alpha + E$) with parameter count ($d$) exponent of $\alpha = 0.31$, close to the $0.34$ in Hoffmann *et al.* (2022). Because the tokens are held constant across all scales, the second loss term ($B/T^\beta$, for $T$ tokens and $B$ and $\beta$ fit parameters in Hoffmann *et al.* (2022)) is the same for all models and hence included in the fit constant ($E = 1.31$) here.

# I The Maximal Update Parameterization

The maximal update parameterization (muP) enables scale-invariant neural network training by prescribing width-dependent initialization and learning rate scalings. A modern interpretation of muP is presented in terms of spectral norms and linear algebra Yang *et al.* (2023). Let $W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ denote the weight matrix for layer $\ell$. We require that a vector $h_{\ell-1}$ of norm $\Theta(\sqrt{n_{\ell-1}})$ maps to $h_\ell := W_\ell h_{\ell-1}$ with norm $\Theta(\sqrt{n_\ell})$. This is ensured if the spectral norm satisfies:

$$\|W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right),$$

which can be understood as maintaining vector norms throughout the network. The spectral norm controls the largest singular value of each layer, hence for an input with typical norm $\sqrt{n_0}$, each layer will in turn have typical norm $\sqrt{n_\ell}$, as expected from a vector of size $n_\ell$ whose entries are $\Theta(1)$. This condition, interpretable as $\sqrt{\text{fan\_out}/\text{fan\_in}}$, enforces stable propagation of activations and gradients. Under gradient descent, updates $\Delta W_\ell$ with the same scaling ensure that $\Delta h_\ell = \Delta W_\ell h_{\ell-1} + \ldots$ remains order-one.

Contrast this approach with standard arguments (e.g., Yang (2019)) which enforce this kind of scale invariance by ensuring stable forward and backward dynamics by requiring:

$$\mathbb{E}\big[W_\ell^{i_1 j_1} W_\ell^{i_2 j_2}\big] \sim \frac{1}{n^p}, \qquad \eta \sim \frac{1}{n^q}, \tag{7}$$

where $n$ is the layer width, $i_1, i_2, j_1, j_2$ are indices and $p, q$ are chosen to avoid exploding or vanishing signals. That this can be accomplished is non-obvious, and is the subject of a substantial body of literature Yang (2019); Yaida (2022); Halverson (2024). While generally less interpretable, this description of muP is convenient for two reasons. First, it generally makes clear the effect of re-parameterizations. Because the learning dynamics have three free parameters (initialization variance, learning rate and initialization scale) but only two constraints (Eq. 7), one degree of freedom remains to choose convenient parameterizations (e.g., to allow weight tying between the embedding and unembedding). The spectral and dynamical conditions we have mentioned can also be shown to agree: the spectral condition guarantees forward and backward stability and matches earlier $mu$P tables Yang (2019); Yang *et al.* (2022), and specifically Table 5. Second, this analysis allows a simple accounting for the possible sources of error in using muP, which we will look at in the following section.

# J Hyperparameter Drift under MuP Scaling

While muP yields approximate scale-invariance, finite-width effects introduce predictable hyperparameter drift. Let $f(x, n)$ be the output of a width-$n$ network under hyperparameter $x$ (e.g., learning rate), and $f_0(x)$ its infinite-width limit. Then:

$$f(x, n) = f_0(x) + \frac{f_1(x)}{n} + O\left(\frac{1}{n^2}\right).$$

Minimizing a loss $\mathcal{L}(f(x, n))$ leads to an optimum $x^\star(n)$ that drifts from its infinite-width counterpart $x^\star$:

$$x^\star(n) = x^\star - \frac{\alpha}{n} + O\left(\frac{1}{n^2}\right),$$

where $\alpha$ depends on $f_0$, $f_1$, and $\mathcal{L}$. Expanding the loss:

$$\ell(x, n) = \mathcal{L}(f(x, n)) = \ell_0(x) + \frac{1}{n}\ell_1(x) + O\left(\frac{1}{n^2}\right), \quad \ell_1(x) = \mathcal{L}'(f_0(x))f_1(x).$$

Imposing optimality yields:

$$\alpha = -\frac{\ell_1'(x^\star)}{\ell_0''(x^\star)} = -\frac{\mathcal{L}''(f_0)\, f_0'\, f_1 + \mathcal{L}'(f_0)\, f_1'}{\mathcal{L}'(f_0)\, f_0'' + \mathcal{L}''(f_0)\, [f_0']^2}.$$

At an extremum where one term in the denominator vanishes, we recover simplified drift laws:

$$\alpha = \begin{cases} \dfrac{f_1(x^\star)}{f_0'(x^\star)}, & \text{if } \mathcal{L}'(f_0(x^\star)) = 0, \\ \dfrac{f_1'(x^\star)}{f_0''(x^\star)}, & \text{if } f_0'(x^\star) = 0. \end{cases}$$

Thus, hyperparameter optima shift by $O(1/n)$ due to finite-width corrections, with the scale and direction of the drift governed by the structure of $f_0$ and $f_1$. This formalizes and quantifies the empirical observation that minima shift during width scaling (e.g., see Appendix of Everett *et al.* (2024)), and provides the key error source in muP-based hyperparameter transfer.