Benchmarking Optimizers for Large Language Model Pretraining

Anonymous Author(s)
Affiliation
Address
email

Abstract

The recent development of Large Language Models (LLMs) has been accompanied by an effervescence of novel ideas and methods to better optimize the loss of deep learning models. Claims from those methods are myriad: from faster convergence to removing reliance on certain hyperparameters. However, the diverse experimental protocols used to validate these claims make direct comparisons between methods challenging. This study presents a comprehensive evaluation of recent optimization techniques across standardized LLM pre-training scenarios, systematically varying model size, batch size, and training duration. Through careful tuning of each method, we provide guidance to practitioners on which optimizer is best suited for each scenario. For researchers, our work highlights promising directions for future optimization research. Finally, by releasing our code and making all experiments fully reproducible, we hope our efforts can help the development and rigorous benchmarking of future methods.

1 Introduction

Over the past five years, Large Language Models (LLMs) [15, 59, 22, 48] have shown growth in performance and size, demonstrating proficiency in various downstream tasks [80, 7, 85]. The success of LLM pretraining hinges on three key pillars: high-quality data [65, 44], architectural innovations [31, 15], and scalable optimization techniques.

Among these, the choice of optimizer has remained notably consistent in recent years, with `Adam(W)` [38, 50] dominating deep learning for nearly a decade. However, recent advances [33, 47, 84, 62, 66, 17] challenge this status quo, offering alternatives that surpass `AdamW` in speed, communication efficiency [1] or final downstream performance on various benchmarks [12, 37], particularly for autoregressive language modeling [70]. Despite these innovations, current benchmarks and ablation studies [96, 34] remain narrow in scope, often examining only isolated aspects of optimizer design. This lack of systematic comparison makes it difficult to obtain trustworthy insights for practitioners, or identify the next promising research directions.

In this work, our goal to revisit the problem of benchmarking optimizers for LLM pretraining. We do so through standardized experiments which vary important parameters such as batch size, model size, and the number of training iterations. This allows us to formulate an up-to-date list of best-performing methods for the community of researchers and practitioners. We demonstrate the efficiency of each considered method through careful tuning, and present insightful ablations along the way. Furthermore, we provide a set of best practices for LLM pretraining that are applicable regardless of the optimizer chosen.

We summarize our contributions as follows:

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

(Contribution 1) We conduct the first large-scale, controlled benchmark of 11 different optimization methods across diverse LLM training scenarios. A fair comparison is ensured by precise accounting for compute costs, and extensive hyperparameter tuning. We identify optimal optimizer choices in several relevant training regimes, for both dense and MoE architectures.

(Contribution 2) We perform comprehensive ablations of critical training hyperparameters—including warmup duration, initialization schemes, gradient clipping, final learning rates, and learning rate scheduler choices—providing actionable insights for optimizing LLM training in practice.

(Contribution 3) We open-source our full benchmarking toolkit, including training scripts, evaluation pipelines, and hyperparameter configurations, to enable reproducible research and facilitate future optimizer development.

For practitioners, our work provides an evidence-based answer to the burning question: "*Is `Adam` still the most effective optimizer in the age of LLMs, or can we achieve better performance at scale with novel optimizers?*". **For researchers**, our work delivers a unified benchmarking framework for LLM pretraining, along with extensive ablation studies which systematically evaluate both popular and overlooked optimizer designs—revealing previously unexplored tradeoffs between efficiency, stability, and final model performance. Overall, our findings not only challenge long-held assumptions about optimizer selection but also establish a foundation for future advances in large-scale model training. By bridging the gap between theoretical innovation and practical deployment, this work aims to accelerate progress in both research and industry applications of LLM training.

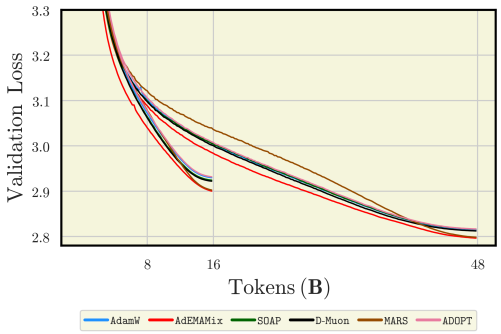


Figure 1: **A comparison of leading optimizers, for training a 720M parameter LLM.**

2 Background & Related Work

Optimizers. While computer vision models often show comparable performance between `SGD` [72] and `AdamW` [94], the landscape differs dramatically in LLM training. Recent work [95] demonstrates that adaptive methods like `AdamW` provide substantially better optimization characteristics for transformer-based language models. The question of why `AdamW` works so well has been a long-standing topic of research [2, 60, 93, 43, 41]. Modern methods often inherit `AdamW`'s core ideas in their structure, such as `ADOPT` [83] and `AdEMAMix` [62]. `ADOPT` has been motivated by solving long-standing convergence issues in `AdamW`. By normalizing the second-order moment prior to the momentum update, they eliminate the non-convergence issues of `AdamW` on smooth non-convex functions. Meanwhile `AdEMAMix` extends `AdamW` with an additional slower momentum buffer, i.e. a slower exponential moving average (EMA), which allows the use of much larger momentum values, accelerating convergence.

One interpretation of `AdamW`'s effectiveness lies in its sign-based update [42]: without the exponential moving average (EMA), `AdamW` resembles `signSGD` [6]. Recent work [96, 36] has shown that `Signum` (`signSGD` with momentum), can perform comparably to `AdamW`. Earlier, the community also discussed `Lion` [9], a method with a similar sign-based structure. `Signum` and `Lion` offer memory benefits due to the use of only a single instead of Adam's two buffers for optimizer states.

Another family of methods stems from `AdamW`'s approximate second-order structure, where the diagonal of the Fisher information matrix or other preconditioning approaches [52, 24] are used as the second moment estimate. This idea has given rise to `Sophia` [46], `SOAP` [84], and, to some extent, `Muon` [33].

The parameter-free concept [61] has led to the development of `Schedule-Free AdamW` (`SF-AdamW`) [17] and `Prodigy` [54]. These optimizers do not require a decreasing learning rate schedule, making them relevant for continual training. Last but not least, `MARS` [88], builds upon this line of research and incorporates a variance reduction mechanism in its update rule.

Benchmarks. To a large extent, the benchmarking setup determines the final conclusions. Some benchmarks are designed for short speedruns in terms of training or validation loss [32], while others focus on a downstream target metric after training [96, 12, 76]. Methods that perform well in short speedruns might not be optimal for longer training horizons as in real LLM training runs (see Figure 3). *"But what constitutes a sufficiently long horizon?" "What should be the compute budget for LLM training?"* These are questions explored by scaling laws [35]. Early benchmarks for optimizers and other ablation studies often rely on Chinchilla scaling laws [26] with a ratio of roughly 20 tokens per parameter (TPR) needed for pretraining. However, recent research [69, 74] argues that this is far from sufficient for production-ready models.

Another important issue is the choice of loss function. Recent setups have been using an auxiliary z-loss [86, 11] in addition to cross-entropy, which requires further investigation. We believe this choice is influenced by the use of the OLMo [58] codebase, which we also address in our work.

Additionally, we found that previous setups for comparing optimizers do not align with recent best practices regarding weight decay, learning rate decay, and overall hyperparameter tuning. All of these questions are revisited in our work.

3 Experimental Setup

Notations. We use the following notations. Let γ be the learning rate, λ the weight decay coefficient, and T the total number of iterations. Momentum-related parameters are represented by the symbol β.

Models & Data. For most experiments, we use a Llama-like transformer [48] architecture, including SwiGLU activations [77], RMSNorm [91], and RoPE embeddings [81]. We experiment with four sizes of models: $124\mathbf{M}$, $210\mathbf{M}$, $583\mathbf{M}$, $720\mathbf{M}$. We train on a $100B$ tokens[1] subset of FineWeb [64]. It consists of a cleaned and deduplicated corpus for LLM pretraining, which we tokenize using the GPT-2 tokenizer prior to splitting into train and validation sequences. MoE setup described in Appendix D.

Iterations & Batch size. Throughout our experiments, we use a sequence length of 512 tokens. For clarity, we often report the batch size in tokens by writing *Batch size×sequence length*. For the $124\mathbf{M}$ model, we use batch sizes of $32 \times 512 = 16\mathbf{k}$, $256 \times 512 = 131\mathbf{k}$, and $512 \times 512 = 262\mathbf{k}$ tokens; for the $210\mathbf{M}$ model, we use a batch size of $256 \times 512 = 131\mathbf{k}$; and for $583\mathbf{M}$ model, we leverage the batch sizes of $1024 \times 512 = 524\mathbf{k}$ and $3936 \times 512 = 2\mathbf{M}$ tokens. Depending on the model size, we vary the number of iterations — also measured in tokens for compatibility with scaling laws and to accommodate different batch size settings. We train $124\mathbf{M}$ and $210\mathbf{M}$ models for equal durations of $\{1, 2.1, 4.2, 6.3, 8.4, 16.8\}\mathbf{B}$ tokens. This corresponds to $T \in \{64, 128, 256, 384, 512, 1024\}\mathbf{k}$ iterations for a batch size of 32, and $T \in \{8, 16, 32, 48, 64, 128\}\mathbf{k}$ iterations for a batch size of 256. For $583\mathbf{M}$ models, we train on $\{13, 32\}\mathbf{B}$ tokens, corresponding to $T \in \{6.5, 16\}\mathbf{k}$ iterations, resp. $T \in \{25, 61.5\}\mathbf{k}$ iterations, for a batch size of 3936, resp. 1024. In the setup with $720\mathbf{M}$ model, we have $T \in \{8, 16, 48\}\mathbf{k}$ iterations for a batch size of $1\mathbf{M}$ tokens. Thus, for all model scales, we include both Chinchilla optimal lengths of training and beyond. More details are available in Appendix C.

Loss. We train using the classical cross-entropy next token prediction loss. Some prior works introducing optimizers use a z-loss in addition to cross-entropy [30, 11, 86, 84, 96]. We found that this has little impact and, therefore, do not use z-loss. An ablation showing results with and without z-loss is in the appendix.

Hyperparameter Tuning. Training LLMs is a computationally intensive task. As a guidance, practitioners often rely on insights gathered at lower scales, scaling laws, and other rules [87, 18]. It is also commonplace to run experiments for only a shorter duration of training, as a way to test certain hyperparameters prior to extending the training horizon to more iterations. Because a full grid search over every hyperparameter, for each setting and optimizer, would be too costly, we resort to a similar approach. More precisely, for each model size, batch size, and optimizer, we tune optimization hyperparameters extensively for a number of training tokens which is near-Chinchilla optimal. We then keep those hyperparameters when we increase the number of iterations. While we found that the sensitivity to several hyperparameters can change as we increase the training horizon, we found this approach simple and yet effective. The hyperparameters being considered depend on the

[1] https://huggingface.co/datasets/HuggingFaceFW/fineweb

optimizer. We proceeded from small to large model scale, and used insights gathered at smaller scales to guide the hyperparameter search at larger scales. Our hyperparameter sweeps are summarized in Appendix D. We present the clarifications regarding the connection between the number of iterations and tokens for different batch size settings as well as the Chinchilla optimal length of training for our models in Tables 3 and 5. As learning rate schedulers, we compare cosine [49], linear and warmup-stable-decay (WSD) [27, 90, 28]. Unless specified, we use a cosine scheduler. Results with WSD and linear schedulers are discussed in Section 4. Recent works also emphasize the importance of sufficiently decaying the learning rate [4, 75, 28]. As such, we take care to decay to $0.01 \times \gamma$ instead of the often used $0.1 \times \gamma$. To give an idea of how much effort was put into tuning each method, across all model sizes, batches and iterations, we trained a total of 2400 models, and have spent roughly 30000 GPU hours.

Optimizers. Here is a list of the optimizers we considered in our work. For each algorithm, we write in parentheses the optimizer-specific hyperparameters we tuned: `AdamW`(β_1, β_2), `SOAP`(β_1, β_2) and preconditioning frequency, `Lion`(β_1, β_2), `MARS`(η, β_1, β_2) and Newton-Schulz hyperparameters, `ADOPT`(β_1, β_2), `Signum`(β), `Prodigy`(β_1, β_2), `SF-AdamW`(β_1, β_2), `Muon`$(\gamma^{\texttt{M}}, \beta, \beta_1, \beta_2)$, `Sophia`$(\rho, \beta_1, \beta_2)$, `AdEMAMix`$(\beta_1, \beta_2, \beta_3, \alpha)$. When an optimizer has several momentum variants e.g. Nesterov [57] or Polyak [67], we try both. In addition, we tune the learning rate γ extensively for all methods. We also try different gradient clipping, warmup steps, and weight-decay values. A summary of the hyperparameters tested and selected for each model size is in Appendix D. All the optimizers are described in depth in Appendix A.

4 Results

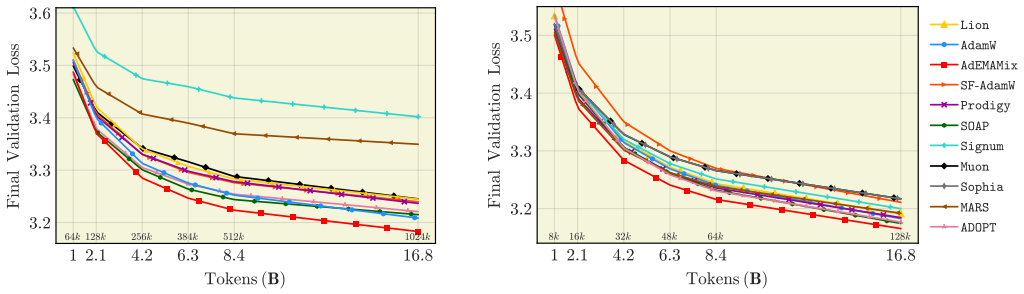
We structure our story starting with smaller models and batch sizes, and gradually scaling up to larger configurations. In some instances, we complement the core benchmarking results with additional ablations and possible best-practices.

4.1 Benchmarking at Small Scale: Training Models of 124M Parameters

Using "small" batches. We first report results when using batches of 32×512 tokens in Figure 3. We tune hyperparameters by training for $2.1\mathbf{B}$ tokens ($128\mathbf{k}$ iterations) and then keep those hyperparameters for all other training durations. The best hyperparameters are reported in Appendix D.1. We observe how, for the smallest number of iterations we considered ($1\mathbf{B}$ tokens $\equiv 64\mathbf{k}$), `SOAP`, `ADOPT` and `AdEMAMix` all outperform `AdamW`, with `SOAP` being the best. As we increase the number of iterations, `AdEMAMix` takes the lead while `AdamW` closes the gap with both `ADOPT` and `SOAP`. A sign-based methods such as `Lion` and `Signum` are expected to perform poorly when the batch size is small. Intuitively, this is due to the $\text{sign}(\cdot)$ operator being sensitive to gradient noise. As described in its original paper, `MARS` also performs poorly when the batch size is small. We found `Prodigy`, `Muon` and `SF-AdamW` to underperform in this setting compared to `AdamW`. On this scale, `Prodigy` suffers from the lack of bias correction of the learning rate, as well as being sensitive to (β_1, β_2) (see Figure 17.

Using "large" batches. We now report results when using batches of 256×512 tokens — $8\times$ larger than for our small batch setting. Results in Figure 2 show how `Signum`, `Mars`, `Lion`, `Prodigy` greatly benefit from the increased batch size. Remarkably, we observe that the `Prodigy` method scales similarly to `AdamW`. We emphasize the possible community interest in this algorithm as its EMA `Prodigy` adaptively emulates the learning rate behaviour. For a small number of iterations (e.g. $T \in \{8\mathbf{k}, 16\mathbf{k}\}$ corresponding to $1\mathbf{B}$ and $2\mathbf{B}$ tokens), all methods outperform `AdamW` except for `SF-AdamW` and `Sophia`. As we increase the number of iterations `ADOPT`, `SOAP`, and `AdEMAMix` take the lead. In particular, `AdEMAMix` has a consistent lead over other methods. While we anticipated—in accordance with Vyas et al.[84]—that `SOAP` would greatly benefit from the larger batch size, its behavior remains relatively consistent compared to our previous small batch setting.

Stability across training horizons. As mentioned in Section 3, we tune hyperparameters training on $2.1\mathbf{B}$ tokens and keep those hyperparameters when extending the training horizon. In Figure 3 we study whether it is possible to find better parameters for `AdamW`, `SOAP`, and `AdEMAMix`. When training on $16.8\mathbf{B}$ tokens, we see it is beneficial to increase the β_3 from 0.999 to 0.9999. Without this improvement, `SOAP` ends up matching the performances of `AdEMAMix` when extending the training horizon further to $33.6\mathbf{B}$ tokens ($\equiv 256\mathbf{k}$ iterations). In our experiments, $\beta_3 = 0.999$ is only better



(a) Batch size 32×512 tokens.

(b) Batch size 256×512 tokens.

Figure 2: **Ranking of optimizers for 124M models with small and large batch sizes.** In both **(a)** and **(b)**, we show the *final* validation loss for different training durations, corresponding to different numbers of tokens. Above each token number, we write the number of training iterations corresponding. In **(a)**, we use a "small" batch size of 32×512 tokens. In **(b)**, we use a larger batch size of 256×512 tokens.

than $\beta_3 = 0.9999$ when the number of training iterations is less than 32k. This matches observation from [62], which recommends reducing β_3 when training for fewer iterations. We also test whether the learning rate γ changes as we increase the number of tokens/iterations. In Figure 5, we run a sweep over γ when training for 16.8B tokens. While for most methods, the best γ obtained in the previous sweep remains optimal, this is not the case for `SOAP` and `SF-AdamW`, which can benefit from a larger $\gamma = 0.002$.

WSD vs. cosine & linear γ-schedulers. Learning rate schedulers received a lot of attention recently [79, 28]. We conducted a series of experiments comparing WSD [27, 90] and linear with cosine [49] learning rate schedulers. Surprisingly, the performance gap between these two schedulers observed in Figure 25 is often significant[2] for benchmarking optimizers. Consequently, we decided to adopt the cosine scheduler for all further experiments.

Decaying γ sufficiently. In Figure 8 we show the impact of decaying more or less the learning rate $\gamma^{(t)}$. From $\gamma = 10^{-3}$ we train models using cosine decay down to $\gamma_{\text{end}} \in \{10^{-4}, 10^{-5}, \dots, 10^{-9}\}$. We found that decaying the learning rate sufficiently matters. In particular, the often use rule consisting in decaying to $0.1 \times \gamma$ is suboptimal. This agrees with the recent works [28, 75, 4]. Building on this findings, we consistently use cosine decay down to $0.01 \times \gamma$.

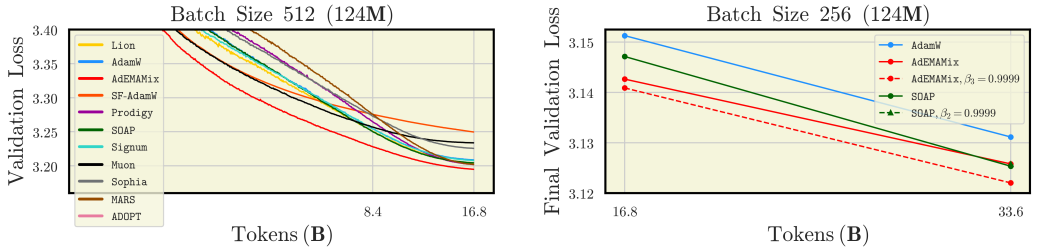
Takeaway 1. After the experiment in the small-batch setting, we conclude that: (i) `AdEMAMix` scales in the best manner with the number of iterations, `SOAP` underperforms `AdamW` when the length of training increases. `ADOPT` and `Prodigy` show almost equal performance across all training durations. Sign-based methods, predictably, underperform when the batch size is small, but what is interesting, is that `Sophia` diverges at all, even if trains with sufficiently small learning rate.

Increasing the batch size further. We also run an experiment with batches of $512 \times 512 = 262$k tokens, training for 64k iterations. Results in Figure 3 show mostly consistent results. Noticeably `MARS` becomes the second best performing method behind `AdEMAMix`, followed closely by `Prodigy`, `Lion`, and `SOAP`. Interestingly, `Signum` performs comparably to `AdamW`.

Takeaway 2. Taking into consideration large batch size setting, we found that many methods, once properly tuned, can show a remarkable performance compared to `AdamW` and also outperform it.

Weight decay ablation. As recent frameworks for LLM pretraining or ablation studies omit weight decay as a default non-zero hyperparameter, some setups even mislead by not incorporating weight decay in their experiments. In this work, we demonstrate the importance of weight decay and its

[2]We emphasize that the difference between the two schedulers is generally less than 5% of the total compute spent. However, this still represents a significant gap in our benchmarking setup, e.g., `SF-AdamW` may outperform `AdamW` in some settings (see Figure 25).

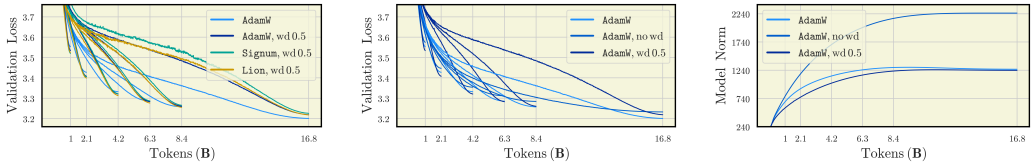


(a) Scaling batch size. (b) Scaling number of tokens.

Figure 3: **Our results demonstrate that (a)**: scaling the batch size significantly improves `MARS`, `Signum`, `Lion` and `Prodigy` making them as good as `AdamW` even for a long training for 16.8**B** tokens. Which was not the case in Figure 2 (b), where we still observed a significant gap in performance; and **(b)**: indeed, with scaling of the number of iterations, the gap between `SOAP` and `AdEMAMix` narrow and, finally, increases. But, on the other hand, with increase of the `AdEMAMix` β_3 parameter, the performance gap with `SOAP` reappears.

impact across different optimizers. Surprisingly, increasing weight decay while keeping the learning rate constant proves to be an effective technique for training on shorter horizons. This approach is so effective that methods like `Signum` and `Lion` with high weight decay significantly outperform `AdamW` without weight decay (see Figure 4). Implementation details also warrant attention. Coupled weight decay is still used in some settings, including the PyTorch [63] optimizer implementations. Notably, the popular implementation of `Signum` becomes ineffective when weight decay is applied. Highlighting this oversight for the community, we contribute by demonstrating our implementation of `Signum` (Algorithm 6) with decoupled weight decay. The influence of weight decay on model weights is intriguing. As is known, model weights typically grow during training, but weight decay, by modifying the optimized function, significantly reduces the growth of the model's parameter norm. Such ablations of weight decay are also of interest to the community [13, 40].

Regarding the ablation of weight decay for optimizers, we again select the best setup for each and conduct a sweep over weight decay values. Our results are presented in Figure 4 and in Figure 23.



(a) (b) (c)

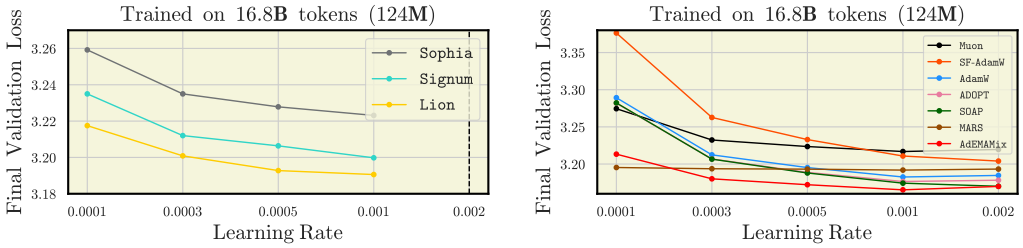
Figure 4: **Larger weight decay achieves significantly better results when training on fewer tokens.** In **(a)** we observe that runs of `AdamW`, `Signum`, and `Lion` with the large weight decay of 0.5 consistently outperform the baseline `AdamW` with weight decay of 0.1 for all training durations except for the last one. Notably, `Signum` and `Lion` with large weight decay perform even better than `AdamW` with the same learning rate. In **(b)**, we also consider a setting without weight decay. We observe that this is suboptimal not only for `AdamW`, but also for the majority of other optimizers (see Appendix E.2), while the typical weight decay of 0.1 remains the best for large training durations. Importantly, in **(c)**, we ablate the impact od weight decay on the model's ℓ_2 norm.

With our weight decay ablation, we are ready to provide one more insight.

> **Takeaway 3.** The use of weight decay, particularly a large decoupled weight decay term, can significantly impact the final loss value and optimizer behavior. However, for extended training horizons, a moderate, non-zero weight decay proves to be a robust option.

Learning rate sensitivity. Since we tune optimizers at a smaller scale and then extrapolate, we pose the question whether the best learning rate we have found so far transfers to the larger training duration. To verify this, we run 124**M** model on 16.8**B** tokens in 256×512 batch size setting,

sweeping the learning rate across five typical values: $\{1e^{-4}, 3e^{-4}, 5e^{-4}, 1e^{-3}, 2e^{-3}\}$. The best learning rate for each method at the moment of hyperparameter tuning on near Chinchilla-optimal $2.1\mathbf{B}$ training duration we report in Appendix D.1. A summary of our results for larger number of tokens is provided in Figure 5 and detailed results of the sweep are presented in Appendix E.2.



(a) (b)

Figure 5: **Optimal learning rate stability across optimizers.** The optimal learning rate determined during tuning on $2.1\mathbf{B}$ tokens remains consistent after a learning rate sweep on $16.8\mathbf{B}$ tokens for most optimizers. In **(a)**, we observe that sign-based methods and similar to them `Sophia` diverge with increasing learning rate. Interestingly, in **(b)**, `SF-AdamW` and `SOAP` demonstrate the best performance with a large learning rate of 0.002. In our work, we further show that it is possible to increase the learning rate even more for such methods.

Warmup ablation. Another important ingredient of the pretraining is learning-rate warmup in the initial phase of training. Recent studies have explored the necessity of warmup in modern deep learning, with some investigating its elimination [39] and others ablating it to improve model performance and stability [92]. We focus on the latter, examining how warmup affects optimizer setup and whether it can significantly enhance performance. For each optimizer's best configuration, we vary warmup across three values: $\{0.27, 1, 4.2\}\mathbf{B}$ tokens, which corresponds to $\{2, 8, 32\}\mathbf{k}$ iterations. Our choice of the largest warmup value is inspired by [92]. We describe this experiment in Appendix E.2. Mainly, we observe that `Signum` and `SF-AdamW` perform better with a larger warmup of $8\mathbf{k}$ steps when training on $16.8\mathbf{B}$ tokens. We also ablate the claim from [92] that a warmup of 25% of the Chinchilla optimal duration is the best. However, our findings contradict this assertion (see Figure 19). We show that a moderate values of the warmup, generally, is better, however, different optimizers could prefer different number of warmup steps. As such, `SF-AdamW`, `Sophia`, `Signum` prefer larger warmup, which is clearly depicted in Figure 6.

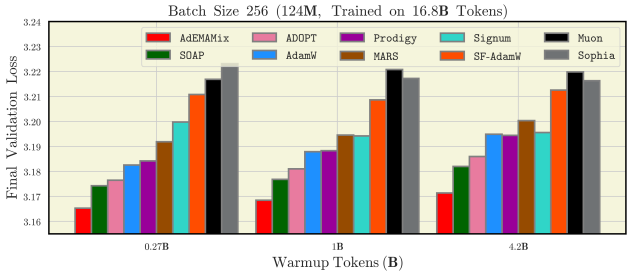
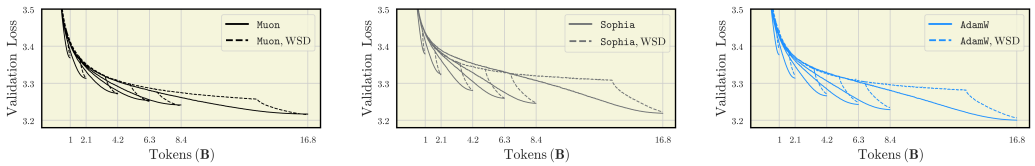


Figure 6: **Warmup ablation.** We report the final validation loss on the FineWeb dataset for $124\mathbf{M}$ model trained on the batch size of 256. We sweep over the batch sizes of $\{1.56\%, 6.25\%, 25\%\}$ of the length of training, which corresponds to $\{2000, 8000, 32000\}\mathbf{k}$ iterations, respectively.

Cosine vs WSD. At the outset of our study, we indicated a preference for the cosine scheduler over WSD. In this section, we provide a more detailed ablation of this choice. Having optimally tuned the cosine scheduler for each optimizer, we replicate the setup of [28], which allows us to avoid adjusting additional hyperparameters. Our findings, which demonstrate the superiority of the cosine scheduler across various optimization methods, are presented in Figure 7, and in the Appendix Figures 25 and 26. These results not only validate our initial preference but also provide insights into the interaction between learning rate schedules and different optimizers in large-scale language model training.



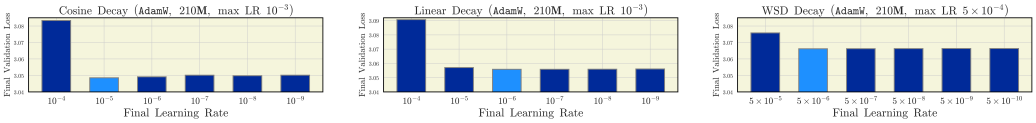
(a) (b) (c)

Figure 7: **Comparisons between WSD and cosine scheduler.** Notably, WSD and cosine scheduler behave differently with respect to optimizer. In **(a)**, the `Muon` optimizer shows a preference for WSD across most training durations. `Sophia` exhibits an almost perfect match between both schedulers. However, for `AdamW`, along with the majority of other optimizers studied (see Figure 26), we get a better performance with cosine. We also report a detailed comparison with linear scheduler in Appendix E.2.

4.2 Benchmarking at medium scale: Training Models of 210M Parameters

In this section, we verify if our selected hyperparameters from smaller 124M allow accurate transfer to a slightly larger model. We point out that the most important hyperparameters to be sweeped are learning rate and gradient clipping. Regarding the learning rate, we observe that it only becomes a sensitive choice for sign-based methods, while the optimal hyperparameters for `AdamW` remain the same.

Results with a batch size of 256×512. Results provided in the Appendix in Figure 21 are consistent with those obtained training 124M models with large batches.



(a) (b) (c)

Figure 8: **Decaying the learning rate down to $0.01 \times \gamma_{\max}$ and beyond, instead of only to 10%** We observe a common pattern for different schedulers that decreasing the learning rate to moderate 10^{-2} value is a better choice than decreasing it down to zero. Interestingly, the linear learning rate scheduler for models at a given scale, requires $0.001 \times \gamma_{\max}$.

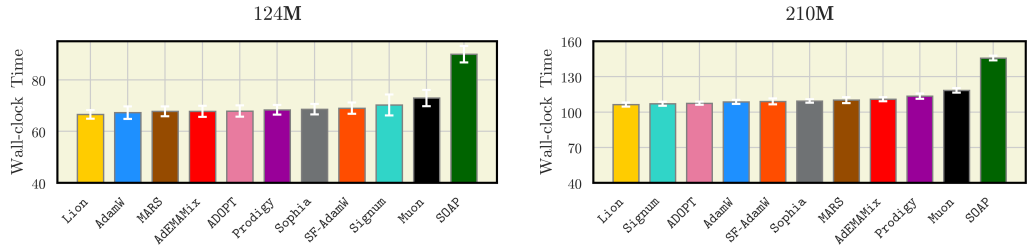


Figure 9: **Wall-clock time comparison.** After conducting experiments for 124M and 210M models, we are ready to present the wall-time comparison for each methods. For this purposes, we use a single GPU, and run each optimizer for 100 iterations on a small batch size without gradient accumulation and `torch.compile`. We report the wall-clock time per 100 iterations. We observe that all methods take the roughly the same time or very close time to complete 100 iterations, with the exception of `Muon` and `SOAP`. In addition, we point out that `SOAP`'s runtime exhibits a non-linear dependence on the model size, due to its preconditioner matrices operations which are fast only for certain matrices smaller than a predefined size.

4.3 Scaling Up: Benchmarking models of 583M and 720M Parameters

We pick three methods: AdamW, SOAP, and AdEMAMix, and run experiments with a larger model of 583M parameters, and a large batch size of 2M tokens. The goal being to get closer to one of the

settings described in [84]. We train for 6500 and 16000 iterations, corresponding to 13**B** and 32**B** tokens respectively.

Comparison between our setting and [84]. We found several key differences between our codebase and [84]: (i) we decay the learning rate to $0.01 \times \gamma$ instead of $0.1 \times \gamma$, with γ being the maximum learning rate, (ii) we use typical weight decay values of e.g. 0.1 instead of smaller values such as 0.01 or 0.0001, (iii) we do not use a z-loss in addition to ours. It has been shown recently that properly decaying the learning rate has an important effect on the optimization [4]. We run an ablation to compare both settings and conclude that removing the z-loss and increasing the weight decay to 0.1 improves the results. Results further improve when the learning rate is decayed more. This ablation is shown in Figure 8.

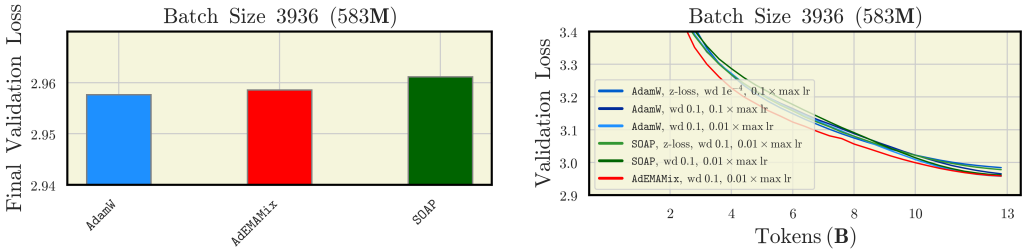


Figure 10: **Results for the** 583M **model.** On the **left**, we show our results when training for 6500 iterations. In this setting, AdamW gives best results, followed by AdEMAMix and then SOAP. This is surprising as it conflicts with findings from [84]. Those results are partly reconciled with the figure on the **right**. And we see that the difference in performance between models trained with and without the z-loss regularizer is quite minor.

5 Extension to MoEs

Setup & Comparison. Besides training dense Llama-like transformers, we also conver a comparison for MoE architectures [78]. Our variant of MoE is based on the Switch-Transformer implementation [20]. We use a classical linear gating with softmax and top-k routing ($k = 2$) and 8 experts. The activation functions remains the same as for the dense base model from Section 3. Such a configuration of the MoE model gives us approximately 520M parameters. We cover additional details in Appendix E.6. In this setting we train with a batch size of 256×512 for $T \in \{42, 336\}$**k** iterations. Again we cover both a Chinchilla-optimal horizon and the beyond. We summarize the results in the following Table 1.

Opt.	42k	336k
AdEMAMix	22.37	18.47
D-Muon	22.67	18.51
ADOPT	22.70	18.58
AdamW	22.85	18.69
Prodigy	22.82	18.78

Opt.	42k	336k
Lion	23.20	18.87
Signum	23.31	19.09
SF-AdamW	23.34	19.13
Sophia	23.41	19.22
MARS	22.73	19.33

Table 1: **Final validation perplexity for MoE training** (↓)**.**

6 Discussion

Our advices on tuning each method. Overall, we validate the already widely used hyperparameters of $\lambda = 0.1$ and $T_{\text{warmup}} \approx 2$**k**. For Lion—as mentioned in [9]—we find that the best value for β_1 is consistently 0.99. The mechanism for Lion seems similar to AdEMAMix, one can imagine that Lion could be better with larger β_1, which would require schedulers. We also pose an interesting observation toward Prodigy: while it may not be so efficient with a super small batch sizes, with scaling of the model size and the batch size it becomes almost as competitive as AdamW. Importantly, Muon and D-Muon performed poorly at a small scale with relatively small batch sizes (32, 256), however, as we see in Figure 1

References

[1] Kwangjun Ahn and Byron Xu. Dion: A communication-efficient optimizer for large models, 2025.

[2] Lukas Balles and Philipp Hennig. Dissecting adam: The sign, magnitude and variance of stochastic gradients, 2020.

[3] C. Bekas, E. Kokiopoulou, and Y. Saad. An estimator for the diagonal of a matrix. *Applied Numerical Mathematics*, 57(11):1214–1229, 2007. Numerical Algorithms, Parallelism and Applications (2).

[4] Shane Bergsma, Nolan Dey, Gurpreet Gosal, Gavia Gray, Daria Soboleva, and Joel Hestness. Straight to zero: Why linearly decaying the learning rate to zero works best for llms, 2025.

[5] Jeremy Bernstein and Laker Newhouse. Old optimizer, new norm: An anthology, 2024.

[6] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: Compressed optimisation for non-convex problems, 2018.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[8] David Edwin Carlson, Edo Collins, Ya-Ping Hsieh, Lawrence Carin, and Volkan Cevher. Preconditioned spectral descent for deep learning. In *Neural Information Processing Systems*, 2015.

[9] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms, 2023.

[10] Savelii Chezhegov, Yaroslav Klyukin, Andrei Semenov, Aleksandr Beznosikov, Alexander Gasnikov, Samuel Horváth, Martin Takáč, and Eduard Gorbunov. Gradient clipping improves adagrad when the noise is heavy-tailed, 2024.

[11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, and Hyung Won. Palm: Scaling language modeling with pathways, 2022.

[12] George E. Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, Juhan Bae, Justin Gilmer, Abel L. Peirson, Bilal Khan, Rohan Anil, Mike Rabbat, Shankar Krishnan, Daniel Snider, Ehsan Amid, Kongtao Chen, Chris J. Maddison, Rakshith Vasudev, Michal Badura, Ankush Garg, and Peter Mattson. Benchmarking Neural Network Training Algorithms, 2023.

[13] Francesco D'Angelo, Maksym Andriushchenko, Aditya Varre, and Nicolas Flammarion. Why do we need weight decay in modern deep learning?, 2024.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[15] DeepSeek-AI. Deepseek-v3 technical report, 2024.

[16] Aaron Defazio and Konstantin Mishchenko. Learning-rate-free learning by d-adaptation, 2023.

[17] Aaron Defazio, Xingyu Alice Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. The road less scheduled, 2024.

[18] Nolan Dey, Quentin Anthony, and Joel Hestness. The practitioner's guide to the maximal update parameterization. https://www.cerebras.ai/blog/the-practitioners-guide-to-the-maximal-update-parameterization, September 2024.

[19] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[20] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[21] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[22] Google Gemini Team. Gemini: A family of highly capable multimodal models, 2024.

[23] Alex Graves. Generating sequences with recurrent neural networks, 2014.

[24] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization, 2018.

[25] Nicholas J. Higham. *Functions of Matrices*. Society for Industrial and Applied Mathematics, 2008.

[26] 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. Training compute-optimal large language models, 2022.

[27] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, Xinrong Zhang, Zheng Leng Thai, Kaihuo Zhang, Chongyi Wang, Yuan Yao, Chenyang Zhao, Jie Zhou, Jie Cai, Zhongwu Zhai, Ning Ding, Chao Jia, Guoyang Zeng, Dahai Li, Zhiyuan Liu, and Maosong Sun. Minicpm: Unveiling the potential of small language models with scalable training strategies, 2024.

[28] Alexander Hägele, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro Von Werra, and Martin Jaggi. Scaling laws and compute-optimal training beyond fixed training durations, 2024.

[29] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization, 2019.

[30] Sami Jaghouar, Jack Min Ong, Manveer Basra, Fares Obeid, Jannik Straube, Michael Keiblinger, Elie Bakouch, Lucas Atkins, Maziyar Panahi, Charles Goddard, Max Ryabinin, and Johannes Hagemann. Intellect-1 technical report, 2024.

[31] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[32] Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977. modded-nanogpt: Speedrunning the nanogpt baseline, 2024.

[33] Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cecista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024.

[34] Jean Kaddour, Oscar Key, Piotr Nawrot, Pasquale Minervini, and Matt J. Kusner. No train no gain: Revisiting efficient training algorithms for transformer-based language models, 2023.

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[36] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signSGD and other gradient compression schemes. In *ICML 2019 - International Conference on Machine Learning*, pages 3252–3261. PMLR, 2019.

[37] Andrej Karpathy. NanoGPT. `https://github.com/karpathy/nanoGPT`, 2022.

[38] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[39] Atli Kosson, Bettina Messmer, and Martin Jaggi. Analyzing & reducing the need for learning rate warmup in gpt training, 2024.

[40] Atli Kosson, Bettina Messmer, and Martin Jaggi. Rotational equilibrium: How weight decay balances learning across neural networks, 2024.

[41] Frederik Kunstner. *Why do machine learning optimizers that work, work?* PhD thesis, University of British Columbia, 2024.

[42] Frederik Kunstner, Jacques Chen, Jonathan Wilder Lavington, and Mark Schmidt. Noise is not the main factor behind the gap between sgd and adam on transformers, but sign descent might be, 2023.

[43] Frederik Kunstner, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. Heavy-tailed class imbalance and why adam outperforms gradient descent on language models, 2024.

[44] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Yitzhak Gadre, Hritik Bansal, Etash Guha, Sedrick Scott Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, 37:14200–14282, 2024.

[45] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[46] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training, 2024.

[47] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for llm training, 2025.

[48] AI @ Meta Llama Team. The llama 3 herd of models, 2024.

[49] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[50] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[51] James Martens. New insights and perspectives on the natural gradient method, 2020.

[52] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[53] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[54] Konstantin Mishchenko and Aaron Defazio. Prodigy: An expeditiously adaptive parameter-free learner, 2024.

[55] A.S. Nemirovskii and Yu.E. Nesterov. Optimal methods of smooth convex minimization. *USSR Computational Mathematics and Mathematical Physics*, 25(2):21–30, 1985.

[56] Yu. Nesterov and V. Shikhman. Quasi-monotone Subgradient Methods for Nonsmooth Convex Minimization. *Journal of Optimization Theory and Applications*, 165(3):917–940, June 2015.

[57] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$, 1983.

[58] Team OLMo. 2 olmo 2 furious, 2024.

[59] OpenAI. Gpt-4 technical report, 2024.

[60] Francesco Orabona. Neural networks (maybe) evolved to make adam the best optimizer, 2020.

[61] Francesco Orabona and Dávid Pál. Open problem: Parameter-free and scale-free online algorithms. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 1659–1664, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

[62] Matteo Pagliardini, Pierre Ablin, and David Grangier. The ademamix optimizer: Better, faster, older, 2024.

[63] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[64] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024.

[65] Guilherme Penedo, Hynek Kydlíček, Vinko Sabolčec, Bettina Messmer, Negar Foroutan, Martin Jaggi, Leandro von Werra, and Thomas Wolf. Fineweb2: A sparkling update with 1000s of languages, December 2024.

[66] Thomas Pethick, Wanyun Xie, Kimon Antonakopoulos, Zhenyu Zhu, Antonio Silveti-Falls, and Volkan Cevher. Training deep learning models with norm-constrained lmos, 2025.

[67] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 1964.

[68] Boris Polyak. New method of stochastic approximation type. *Automation and Remote Control*, 1990, 01 1990.

[69] Tomer Porian, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon. Resolving discrepancies in compute-optimal scaling of language models, 2024.

[70] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[71] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019.

[72] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[73] David Ruppert. Efficient estimations from a slowly convergent robbins-monro process. 1988.

[74] Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws, 2024.

[75] Fabian Schaipp, Alexander Hägele, Adrien Taylor, Umut Simsekli, and Francis Bach. The surprising agreement between convex optimization theory and learning-rate scheduling for large model training, 2025.

[76] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley - benchmarking deep learning optimizers, 2021.

[77] Noam Shazeer. Glu variants improve transformer, 2020.

[78] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[79] Yikang Shen, Matthew Stallone, Mayank Mishra, Gaoyuan Zhang, Shawn Tan, Aditya Prasad, Adriana Meza Soria, David D. Cox, and Rameswar Panda. Power scheduler: A batch size and token number agnostic learning rate scheduler, 2024.

[80] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024.

[81] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[82] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[83] Shohei Taniguchi, Keno Harada, Gouki Minegishi, Yuta Oshima, Seong Cheol Jeong, Go Nagahara, Tomoshi Iiyama, Masahiro Suzuki, Yusuke Iwasawa, and Yutaka Matsuo. Adopt: Modified adam can converge with any β_2 with the optimal rate, 2024.

[84] Nikhil Vyas, Depen Morwani, Rosie Zhao, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam, 2024.

[85] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[86] Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Ce Bian, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, et al. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*, 2023.

[87] Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer, 2022.

[88] Huizhuo Yuan, Yifeng Liu, Shuang Wu, Xun Zhou, and Quanquan Gu. Mars: Unleashing the power of variance reduction for training large models, 2024.

[89] Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[90] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers, 2022.

[91] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019.

[92] Hanlin Zhang, Depen Morwani, Nikhil Vyas, Jingfeng Wu, Difan Zou, Udaya Ghai, Dean Foster, and Sham Kakade. How does critical batch size scale in pre-training?, 2024.

[93] Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank Reddi, Sanjiv Kumar, and Suvrit Sra. Why are adaptive methods good for attention models? *Advances in Neural Information Processing Systems*, 33:15383–15393, 2020.

[94] Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank J Reddi, Sanjiv Kumar, and Suvrit Sra. Why are adaptive methods good for attention models?, 2020.

[95] Yushun Zhang, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. Why transformers need adam: A hessian perspective, 2024.

[96] Rosie Zhao, Depen Morwani, David Brandfonbrener, Nikhil Vyas, and Sham Kakade. Deconstructing what makes a good optimizer for language models. *ICLR*, 2025.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: the contribution of this paper is described accurately in the abstract and introduction.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: we discuss a limitations and mention experiments we have not tried to run

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: this is not a theoretical work

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: we open-source our code

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: we provide our code and the datasets are mentioned clearly

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: we specify all of the important hyperparameters as well as hyperparameter tuning.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: in our large-scale experiments we could not affort so. and we are running all of the experiment with the same seed for generation data splits, etc.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: we provide this in Appendix

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: this paper is consistent with NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: there is no societal impact of the work performed.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: the paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: we cite them and respect, see Sections 1 and 2

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: the paper does not propose new assets.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: our work does not include research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: our work does not include research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: the core development of our work does not involve LLMs.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

Contents

A Optimizers we study

In this section, we describe all considered algorithms, presenting them in a unified formalism. We start with notation and then discuss the algorithms according to their logical grouping:

1. Adam-like methods: AdamW (Algorithm 1), ADOPT (Algorithm 2) and AdEMAMix (Algorithm 3).

2. Sign-based methods: Lion (Algorithm 4), Signum (Algorithms 5 and 6).

3. Approximate second-order optimizers: Muon (Algorithm 8), SOAP (Algorithm 10) and Sophia (Algorithm 11).

4. Learning rate / scheduler-free learning algorithms: Schedule-Free AdamW (Algorithm 12), Prodigy (Algorithm 13).

5. MARS algorithms: (Algorithms 14 to 16).

Notation. In our work, we denote all vectors and matrices using bold symbols, while non-bold symbols represent scalars. Let $\mathcal{L} : \mathcal{D} \to \mathbb{R}$ be an empirical loss function parameterized by $\boldsymbol{x}$, and mapping batch of inputs $\boldsymbol{\xi} \subset \mathcal{D}$ to $\mathbb{R}$. As $\boldsymbol{g} = \nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\xi})$ we denote a stochastic gradient of the loss w.r.t. parameters $\boldsymbol{x}$. For simplicity, we omit $\boldsymbol{x}$ in ∇ and write $\nabla \mathcal{L}(\boldsymbol{x}, \boldsymbol{\xi})$. We use the following standardized notation for specific symbols in our work: batch size — $|\boldsymbol{\xi}|$, learning rate — γ, weight decay — λ, momentum — β, iteration counter t with the total number of iterations — T.

And basic notation for symbols in the algorithms: $\boldsymbol{m}, \boldsymbol{v}$ – are first and second moment estimates, respectively, with their bias corrected versions $\hat{\boldsymbol{m}}, \hat{\boldsymbol{v}}$, and beta parameters — β_1, β_2. We denote the dot product of two vectors $\boldsymbol{z}$, $\boldsymbol{y}$ as $\langle \boldsymbol{z}, \boldsymbol{y} \rangle$, while $\boldsymbol{z} \odot \boldsymbol{y}$ stands for their element-wise product. All division and addition operations in the described algorithms are element-wise.

A.1 AdamW, ADOPT, AdEMAMix

AdamW. Our baseline — Adam(W), has become a de facto optimizer for deep learning, demonstrating impressive performance across diverse domains: from tabular data to diffusion and language models.

The method originated from the ideas of Adagrad [19] and RMSProp [23], which utilize a second moment estimate $\boldsymbol{v}$ in their update rule. However, Adam(W) enhanced this prior scheme by incorporating momentum [55, 82], establishing itself as a state-of-the-art method for a wide range of tasks. All other algorithms we consider also employ a similar, if not identical, momentum scheme.

Another key aspect of AdamW is its decoupled weight decay λ [50], unlike Adam. We use the decoupled weight decay scheme for all methods to ensure consistency and emphasize its importance for optimizer comparison, hyperparameter tuning, and final performance. This is clearly observable, e.g., for Signum (Algorithm 5).

Algorithm 1 AdamW

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}$, $\boldsymbol{v}_0 \leftarrow \boldsymbol{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t$
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t \odot \boldsymbol{g}_t$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t), \quad \hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon} + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

ADOPT. Recently, [83] proposed a modification of Adam, by removing the current gradient $\boldsymbol{g}_t$ from the second moment estimate $\boldsymbol{v}_t$ and changing the order of the momentum update $\boldsymbol{m}_t$ and the normalization by the second moment estimate. As shown in line 8 of Algorithm 2, the parameter update depends only on the previous value of the second moment estimate $\boldsymbol{v}_{t-1}$. The authors analyze the convergence of ADOPT with the following update rule:

$$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \frac{\boldsymbol{g}_t}{\max\{\sqrt{\boldsymbol{v}_{t-1}}, \varepsilon\}},$$
$$\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \boldsymbol{m}_t.$$

However, the practical implementation differs in a few details. To tackle instabilities caused by near-zero gradients during the early stages of training, the authors propose using a clipping on $\boldsymbol{g}_t / \max\{\sqrt{\boldsymbol{v}_{t-1}}, \varepsilon\}$, which we formalize as the `clamp` operation. Given a vector $\boldsymbol{g}$ and a positive scalar c, it is defined as:

$$\texttt{clamp}\left(\boldsymbol{g}, c\right)^{(i)} = \min\left\{\max\left\{g^{(i)}, -c\right\}, c\right\}. \tag{1}$$

Thus, the element-wise `clamp` operation preserves $\boldsymbol{g}_t$ from the division by near-zero values.

The authors theoretically claim that `ADOPT` achieves the optimal convergence bound for non-convex objectives, regardless of the choice of the β_2 parameter. We empirically investigate this claim and observe that, contrary to the theoretical results, there is a significant performance gap for different choices of β_2 in practice.

Algorithm 2 `ADOPT`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}$, $\boldsymbol{v}_0 \leftarrow \nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0) \odot \nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla\mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $c_t \leftarrow t^{1/4}$ ▷ Update clipping value schedule
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\texttt{clamp}\left(\frac{\boldsymbol{g}_t}{\max\{\sqrt{\boldsymbol{v}_{t-1}}, \varepsilon\}}, c_t\right)$
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t\left(\boldsymbol{m}_t + \lambda \boldsymbol{x}_t\right)$ ▷ Update without $\boldsymbol{v}_t$
end for
Return: $\boldsymbol{x}_T$

`AdEMAMix`. Another `Adam`-like optimizer we study is `AdEMAMix` [62]. This work argues that using a single EMA to accumulate past gradients in the first moment estimate $\boldsymbol{m}$ can be sub-optimal, as it cannot simultaneously prioritize both immediate past and older gradients. In Algorithm 3, the authors incorporate two EMAs: one — `Adam`-like EMA for $\boldsymbol{m}$ (fast), and another — a slow EMA $\boldsymbol{m}^{\text{slow}}$ (see `line 7`) with an additional β_3 parameter. In its update rule, the algorithms balances fast and slow EMAs with the constant factor α (see `line 10` of Algorithm 3). This algorithmic design helps `AdEMAMix` benefit from older gradients and results in smoother loss curves during training.

However, to mitigate the effect of early instabilities, the authors use two additional schedulers for α and β_3 – `alpha_scheduler` and `beta_scheduler`, formalized in our work as follows:

$$\texttt{alpha\_scheduler}(t, \alpha, T_\alpha) = \min\left\{\frac{t\alpha}{T_\alpha}, \alpha\right\},$$

$$\texttt{beta\_scheduler}(t, \beta_3, \beta_{\text{start}}, T_{\beta_3}) = \min\left\{\exp\left(\frac{\log(\beta_{\text{start}})\log(\beta_3)}{\left(1-\frac{t}{T_{\beta_3}}\right)\log(\beta_3) + \frac{t}{T_{\beta_3}}\log(\beta_{\text{start}})}\right), \beta_3\right\}.$$

In all experiments, we set $\beta_{\text{start}} = \beta_1$, and the warmup parameters equal to the length of training: $T_\alpha = T_{\beta_3} = T$.

One thing that should be commented — α and β schedulers are seemingly contradict the idea of WSD scheduler. However, setting T_α, T_{β_3} to be longer than the first checkpoint of the WSD does not significantly impact the final performance. Thus, `AdEMAMix` can still be combined with recent findings regarding the WSD scheduler.

A.2 Sign-based methods: `Lion` and `Signum`

Another branch of methods includes sign-based `Lion` and `Signum`. To some extend, one can classify `Adam` as a sign-based method also, but we mention only `Lion` and `Signum` as they explicitly incorporate the `sign` operation in the update rule.

These methods, particularly `Signum`, have been unfairly overlooked in the context of LLM pretraining. However, our results demonstrate that, with sufficiently large batch sizes and at moderate model scales, these optimizers perform on par with `Adam`, and in some cases, even outperform it.

Algorithm 3 `AdEMAMix`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, β_3, β_{start}, α, `beta_scheduler`, `alpha_scheduler`, warmup parameters T_{β_3} and T_α, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{m}_0^{\text{slow}} \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\beta_3(t) \leftarrow \texttt{beta\_scheduler}(t, \beta_3, \beta_{\text{start}}, T_{\beta_3})$, $\alpha(t) \leftarrow \texttt{alpha\_scheduler}(t, \alpha, T_\alpha)$ ▷ Update β_3 and α schedulers
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$
  $\boldsymbol{m}_t^{\text{slow}} \leftarrow \beta_3(t)\boldsymbol{m}_{t-1}^{\text{slow}} + (1-\beta_3(t))\boldsymbol{g}_t$ ▷ Update slow EMA
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1-\beta_1^t)$, $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1-\beta_2^t)$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\frac{\hat{\boldsymbol{m}}_t + \alpha(t)\boldsymbol{m}_t^{\text{slow}}}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon} + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

Lion. And the first method of this kind is `Lion` [9]. This optimizer is symbolically discovered in the program space of first-order optimization primitives. `Lion` updates its EMA of $\boldsymbol{m}$ after updating the parameters and has additional term $(1-\beta_1)\boldsymbol{g}$ whis adds to the momentum. This interpolation $\beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$ (see `line 6` of Algorithm 4) makes the symbolic-discovered idea behind `Lion` similar to the idea of the `AdEMAMix` optimizer.

Algorithm 4 `Lion`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta_2 \boldsymbol{m}_{t-1} + (1-\beta_2)\boldsymbol{g}_t$ ▷ Update EMA of $\boldsymbol{g}_t$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\texttt{sign}\left(\beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t\right) + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

Signum. Another sign-based method, which is the adoptation of `signSGD` — `Signum` [6], or `signSGD` with momentum. This method differs from `Lion` in the interpolation term between EMA of momentum and current gradient, as well as in the `Signum`'s update rule a current EMA is used.

Importantly, while `Signum` is not as widespread for LLM pretraining and stands mostly as a theoretical artifact, recent practitioner's studies also start to use `Signum` for scalable training [96]. Mostly, due to the memory-efficiency of `Signum` compared to `AdamW`.

In this regard, we want to make an important point — many recent PyTorch [63] implementations of the `Signum` optimizer, unlikely, are not suitable for this method, impairing the potential performance from using it.

The main problem of many open-source implementations is a use of decoupled weight decay in the PyTorch implementation of `SGDM` (`SGD` with momentum) [82]. Indeed, with a decoupled weight decay, the update of Algorithm 5 transforms into:

$$\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \texttt{sign}\left(\beta \boldsymbol{m}_{t-1} + (1-\beta)\boldsymbol{g}_t - \lambda(1-\beta)\boldsymbol{g}_t\right),$$

which affects the sign of the update, leading to potentially wrong optimization direction if the weight decay is sufficiently large.

Another popular failure while using `Signum` is incorrectly tractable PyTorch implementation of `SGDM`. It does not include such EMA as `line 5` in Algorithm 5, on the other hand, in PyTorch, the momentum update depends on the dampening parameter τ:

$$\boldsymbol{m}_t \leftarrow \beta \boldsymbol{m}_{t-1} + (1-\tau)\boldsymbol{g}_t,$$

which is zero by default. Therefore, the typical update rule, reflecting the actual `Signum` behavior in practice corresponds to the following update:

$$\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\texttt{sign} \left(\beta \boldsymbol{m}_{t-1} + (1-\tau) \boldsymbol{g}_t \right) + \lambda \boldsymbol{x}_t \right),$$

where the weight decay is decoupled and, consequently, does not affect the `sign`.

$$\boldsymbol{g}_t \leftarrow \boldsymbol{g}_t + \beta \boldsymbol{m}_t,$$

improves `Signum`. Since enabling Nesterov momentum requires zero dampening τ, we revisited the descrition of Algorithm 5, and propose more practical, PyTorch-compatible version of `Signum` in Algorithm 6.

Algorithm 5 `Signum` (basic)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, momentum β.
Initialize: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta \boldsymbol{m}_{t-1} + (1-\beta) \boldsymbol{g}_t$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\texttt{sign} \left(\boldsymbol{m}_t \right) + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

Algorithm 6 `Signum` (our PyTorch variant)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, momentum β.
Initialize: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta \boldsymbol{m}_{t-1} + \boldsymbol{g}_t$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\texttt{sign} \left(\beta \boldsymbol{m}_t + \boldsymbol{g}_t \right) + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

Moreover, to prevent other researchers and practitioners from the possible wrong use of `Signum` and for the reproducibility purposes, we provide our Python code.

Listing 1: `Signum` code skeleton using PyTorch

```
from typing import Dict

import torch


class Signum(torch.optim.Optimizer):
    def __init__(
        self,
        params,
        lr=1e-3,
        momentum=0,
        dampening=0,
        weight_decay=0,
        nesterov=False,
        sign_update=True,
    ):
        if lr < 0.0:
            raise ValueError(f"Invalid learning rate: {lr}")
        if momentum < 0.0:
            raise ValueError(f"Invalid momentum value: {momentum}"
                )
        if weight_decay < 0.0:
            raise ValueError(f"Invalid weight_decay value: {
                weight_decay}")

        defaults = dict(
            lr=lr,
```

```
            momentum=momentum,
            dampening=dampening,
            weight_decay=weight_decay,
            nesterov=nesterov,
            sign_update=sign_update,
        )
        if nesterov and (momentum <= 0 or dampening != 0):
            raise ValueError("Nesterov momentum requires a
                momentum and zero dampening")
        super().__init__(params, defaults)

    def __setstate__(self, state):
        super().__setstate__(state)
        for group in self.param_groups:
            group.setdefault("nesterov", False)

    @torch.no_grad()
    def _init_state(self, example, state=None):
        assert isinstance(example, torch.Tensor)
        assert isinstance(state, Dict) or state is None
        if state is None:
            state = {}
        state["step"] = 0
        state["momentum_buffer"] = torch.clone(example).detach()
        return state

    @torch.no_grad()
    def _compute_update(
        self, grad, state, lr, momentum, nesterov, dampening,
            sign_update, **kwargs
    ):
        if momentum != 0:  # Signum check
            buf = state["momentum_buffer"]
            buf.mul_(momentum).add_(grad, alpha=1 - dampening)

            if nesterov:
                grad = grad.add(buf, alpha=momentum)
            else:
                grad = buf

        if sign_update:
            grad = grad.sign()

        return grad * (-lr)

    @torch.no_grad()
    def step(self, closure=None):
        """Performs a single optimization step.

        Args:
            closure (Callable, optional): A closure that
                reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()
```

```
        for group in self.param_groups:
            for p in group["params"]:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]

                if group["weight_decay"] != 0:
                    p.mul_(1 - group["lr"] * group["weight_decay"
                        ])

                if len(state) == 0:
                    self._init_state(example=p, state=state)
                    if not group["momentum"]:
                        state.pop("momentum_buffer", None)

                state["step"] += 1

                update = self._compute_update(
                    grad,
                    state,
                    group["lr"],
                    group["momentum"],
                    group["nesterov"],
                    group["dampening"],
                    group["sign_update"],
                )

                p.add_(update)

        return loss
```

A.3 `Muon`, `SOAP`, `Sophia`

Next page of the methods covers algorithms that rather aim to use more information about the problem's curvature (`SOAP` [84], `Sophia` [46]) or perform fast updates of matrix parameters involving higher order computations (`Muon` [33]).

Contrary to the chronological order, we discuss them starting from the recent one — `Muon` and end up with `Sophia`.

`Muon`. Specifically designed for the speedrun comparisons, this method surpasses `AdamW` baseline on nanoGPT pretraining benchmark [32]. Claims from the `Muon` extend to faster learning, lower memory usage and better sample-efficiency with a small wall-clock time overhead. However, there are not much to say about the final performance given a particular budget of tokens to train on.

The reason why the `Muon` is a good option for speedrun pretraining lies in its structure — `Muon` treats different layers in different way. One dimensional (1D) parameters, large embedding layers, scalar parameters (such as Layer Norm parameters) and the output layer of LLM (`lm_head`) are optimized by `AdamW`. And all parameters with two or more dimensions, e.g., Multi-Head Attention layers are optimized by Algorithm 7, which we call `MuonNon1D`.

Inspired by `Shampoo`'s preconditioners [24], the authors of `MuonNon1D` incorporated an orthogonalization step to compute the `SVD` transformation of gradient matrix. Before the orthogonalization step, `MuonNon1D` represents `SGD` with Nesterov momentum. To ensure fast orthogonalization procedure, the authors, insiped by [5], use Newton-Schulz procedure [25]. As the number of Newton-Schulz iterations increases, the closer resulting matrix becomes to $\boldsymbol{U}\boldsymbol{V}^\top$ from `SVD` transformation. The authors also mention that `Muon` can be thought of as a second way of smoothing spectral steepest descent [8], with a different set of memory and runtime trade-offs compared to `Shampoo`.

Algorithm 7 `MuonNon1D` (for non-1D parameters)

Input: Initial non-1D parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, momentum β, number of Newton-Schulz iterations T_{NS}, a, b, c coefficients.
Initialize: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta \boldsymbol{m}_{t-1} + \boldsymbol{g}_t$
  $\boldsymbol{g}_t \leftarrow \beta \boldsymbol{m}_t + \boldsymbol{g}_t$ ▷ Practical implementation of Nesterov momentum
  Set: $\boldsymbol{w}_0 \leftarrow \boldsymbol{g}_t / \|\boldsymbol{g}_t\|_F$
  for $n \in [T_{\text{NS}}]$ **do**
    $\boldsymbol{w}_{n+1} \leftarrow a\boldsymbol{w}_n + b\boldsymbol{w}_n\boldsymbol{w}_n^\top + c\left(\boldsymbol{w}_n\boldsymbol{w}_n^\top\right)^2 \boldsymbol{w}_n$ ▷ Newton-Schulz iteration
  end for
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \boldsymbol{w}_{T_{\text{NS}}}$
end for
Return: $\boldsymbol{x}_T$

Algorithm 8 `Muon` (general scheme)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T. `Muon`'s parameters: learning rate $\gamma_t^{\texttt{M}}$, momentum β, number of Newton-Schulz iterations T_{NS}, a, b, c coefficients. `AdamW`'s parameters: learning rate $\gamma_t^{\texttt{A}}$, weight decay λ, β_1, β_2, ε.
for $t \in [T]$ **do** $\boldsymbol{x}_t \in \{\texttt{embeds}, \texttt{scalar\_params}, \texttt{lm\_head}\}$
  $\boldsymbol{x}_t^{\texttt{A}} \leftarrow \boldsymbol{x}_t$
  $\boldsymbol{x}_{t+1}^{\texttt{A}} \leftarrow \texttt{AdamW}\left(\boldsymbol{x}_t^{\texttt{A}}, \gamma_t^{\texttt{A}}, \lambda, \beta_1, \beta_2, \varepsilon, T = 1\right)$ ▷ One iteration of `AdamW`
  $\boldsymbol{x}_t^{\texttt{M}} \leftarrow \boldsymbol{x}_t$
  $\boldsymbol{x}_{t+1}^{\texttt{M}} \leftarrow \texttt{MuonNon1D}\left(\boldsymbol{x}_t^{\texttt{M}}, \gamma_t^{\texttt{M}}, T_{\text{NS}}, \beta, a, b, c, T = 1\right)$ ▷ One iteration of `MuonNon1D`
end for
Return: $\boldsymbol{x}_T^{\texttt{A}}, \boldsymbol{x}_T^{\texttt{M}}$

Importantly, we noticed that the original algorithmic description of `Muon` optimizer, provided in the official repository[3], differs from the actual one, presented in Algorithm 7. In the original code, as well as in our benchmarking, weight decay do not applies to the matrix parameters in the optimizer state of `MuonNon1D`, which means that the only weight decay used during training is `AdamW`'s weight decay. From this perspective, we observe that the gap between the final loss values for runs with 0.1 and 0 weight decay values almost disappears, while the run with 0.5 weight decay becomes the worst, which is not the case for other optimizers. We describe this in our weight decay ablations.

`SOAP`. [84] proposed new, improved modification of `Shampoo` [24]. `SOAP` reduces the computational overhead optimizing only two dimensional layers (2D) via Algorithm 9, while running `AdamW` for 1D layers. At initialization, preconditioners are computed via eigenvector decomposition of the initial gradient matrices $\texttt{eigenbasis}\left(\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)^\top\right)$: $\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)^\top = \boldsymbol{q}\Lambda\boldsymbol{q}^{-1}$, where Λ stands for the diagonal matrix whose diagonal elements are the corresponding eigenvalues. For the rest of the iterations, `SOAPNon1D` performs the `QR` decomposition (see `lines 15, 16` of Algorithm 9) for all 2D layers, which is the main computational part of the method.

A key idea behind the `SOAP` optimizer is:

1. Given the slowly changing coordinate basis provided by eigenvectors $\boldsymbol{l}$ and $\boldsymbol{r}$, `SOAP` updates its second moment estimates in this basis, i.e., it runs `AdamW` in another, rotated space.

2. To update the eigenvectors of $\boldsymbol{l}$ and $\boldsymbol{r}$, `SOAP` runs `QR` decomposition with preconditioning frequency ϕ.

In Algorithm 9, if one would set both $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ to identity, then we would recover `AdamW`.

The overall `SOAP` algorithm can be formalized as follows:

[3] https://github.com/KellerJordan/modded-nanogpt

Algorithm 9 `SOAPNon1D` (for non-1D parameters)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, preconditioning frequency ϕ, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
Initialize preconditioners: $\boldsymbol{q}_l, \boldsymbol{q}_r \leftarrow \texttt{eigenbasis}\left(\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)\nabla\mathcal{L}(\boldsymbol{x}_0, \boldsymbol{\xi}_0)^\top\right)$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla\mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{g}'_t \leftarrow \boldsymbol{q}_l^\top \boldsymbol{g}_t \boldsymbol{q}_r$ ▷ Rotate $\boldsymbol{g}_t$
  $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$
  $\boldsymbol{m}'_t \leftarrow \boldsymbol{q}_l^\top \boldsymbol{m}_t \boldsymbol{q}_r$ ▷ Compute `Adam`'s statistics in rotational space
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}'_t \odot \boldsymbol{g}'_t$
  $\gamma_t \leftarrow \gamma_t \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$ ▷ Optional: use bias correction
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\boldsymbol{q}_l \frac{\boldsymbol{m}'_t}{\sqrt{\boldsymbol{v}_t}+\varepsilon} \boldsymbol{q}_r^\top + \lambda \boldsymbol{x}_t\right)$ ▷ Perform update in original space
  $\boldsymbol{l}_t \leftarrow \beta_2 \boldsymbol{l}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \boldsymbol{g}_t^\top$ ▷ Update preconditioners
  $\boldsymbol{r}_t \leftarrow \beta_2 \boldsymbol{r}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^\top \boldsymbol{g}_t \; t \equiv 1 \pmod{\phi}$
  $\boldsymbol{q}_l \leftarrow \texttt{QR}\left(\boldsymbol{l}_t \boldsymbol{q}_l\right)$
  $\boldsymbol{q}_r \leftarrow \texttt{QR}\left(\boldsymbol{r}_t \boldsymbol{q}_r\right)$
end for
Return: $\boldsymbol{x}_T$

Algorithm 10 `SOAP` (general scheme)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, preconditioning frequency ϕ, ε.
for $t \in [T]$ **do** $\boldsymbol{x}_t \in \{\texttt{embeds}, \texttt{scalar\_params}, \texttt{lm\_head}\}$
  $\boldsymbol{x}_t^{\texttt{A}} \leftarrow \boldsymbol{x}_t$
  $\boldsymbol{x}_{t+1}^{\texttt{A}} \leftarrow \texttt{AdamW}\left(\boldsymbol{x}_t^{\texttt{A}}, \gamma_t, \lambda, \beta_1, \beta_2, \varepsilon, T=1\right)$ ▷ One iteration of `AdamW`
  $\boldsymbol{x}_t^{\texttt{S}} \leftarrow \boldsymbol{x}_t$
  $\boldsymbol{x}_{t+1}^{\texttt{S}} \leftarrow \texttt{SOAPNon1D}\left(\boldsymbol{x}_t^{\texttt{S}}, \gamma_t, \lambda, \beta_1, \beta_2, \varepsilon, T=1\right)$ ▷ One iteration of `SOAPNon1D`
end for
Return: $\boldsymbol{x}_T^{\texttt{A}}, \boldsymbol{x}_T^{\texttt{S}}$

`Sophia`**.** Despite being named as second-order optimizer, `Sophia` [46] performs an update, quite similar to `Adam`'s. It also leverages the diagonal preconditioner $\boldsymbol{h}$, but not the curvature information of the optimization problem, which depends on the non-diagonal terms of the Hessian. One should notice that `Sophia` were introduced with two types of preconditioners — Hutchinson [3] and Gauss-Newton-Bartlett [51]. Since the latter one shows more promising performance, we consider only this type of preconditioner for `Sophia`.

Every ϕ iterations, `Sophia` updates its second moment estimate by computing the gradient $\hat{\boldsymbol{g}}$ of the empirical loss $\mathcal{L}$ given `softmax` of the logits instead of the true logits. Multiplying by the batch size, we obtain $\hat{\boldsymbol{h}}$, after that, `Sophia` updates the EMA of $\hat{\boldsymbol{h}}$.

Importantly, we found out, that algorithmic description of `Sophia` in the original paper differs in minor details from the code implementation[4]. Indeed, the update rule in their work formulates as follows:

$$\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \texttt{clamp}\left(\frac{\boldsymbol{m}_t}{\max\{\rho \boldsymbol{h}_t, \varepsilon\}}, 1\right),$$

where `clamp` is defined as in Equation (1). On the other hand, the code from the official repository suggests:

Listing 2: `Sophia` update skeleton using PyTorch

```
# update step
```

[4]https://github.com/Liuhong99/Sophia

```
step_t += 1

# Perform stepweight decay
param.mul_(1 - lr * weight_decay)

# Decay the first and second moment running average coefficient
exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)

else:
    step_size_neg = -lr

    ratio = (exp_avg.abs() / (rho * bs * hess + 1e-15)).clamp(None
        , 1)
    param.addcmul_(exp_avg.sign(), ratio, value=step_size_neg)
```

Therefore, the actual update of `Sophia` is wrongly tractated in the original paper and should be corrected and equal to the `line 16` of Algorithm 11.

Takeaway 4. *The actual update rule of* `Sophia` *differs from its description in the original paper.*

Algorithm 11 `Sophia`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, estimator frequency ϕ, scaling factor ρ, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{h}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \; t \equiv 1 \pmod{\phi}$
  $p_t \leftarrow \boldsymbol{\xi}_t$ ▷ Obtain logits from batch
  $p_t \leftarrow \texttt{softmax}\,(p_t)$ ▷ Sample from logits
  $\hat{\mathcal{L}}(\boldsymbol{x}_t, \boldsymbol{\xi}_t) \leftarrow p_t$ ▷ Loss, where p_t are labels
  $\hat{\boldsymbol{g}}_t \leftarrow \nabla \hat{\mathcal{L}}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\hat{\boldsymbol{h}}_t \leftarrow |\boldsymbol{\xi}_t| \hat{\boldsymbol{g}}_t \odot \hat{\boldsymbol{g}}_t$
  $\boldsymbol{h}_t \leftarrow \beta_2 \boldsymbol{h}_{t-\phi} + (1-\beta_2)\hat{\boldsymbol{h}}_t$
  $\boldsymbol{h}_t \leftarrow \boldsymbol{h}_{t-1}$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t \left(\texttt{sign}(\boldsymbol{m}_t) \min\left\{ \frac{|\boldsymbol{m}_t|}{\rho \boldsymbol{h}_t + \varepsilon}, 1 \right\} + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

A.4 `Schedule-Free AdamW`, `Prodigy`

In this section, we outline two more players — `Schedule-Free AdamW` [17] and `Prodigy` [54]. Both of them have a promising advantages and require less hyperparameter tuning which paves the road to parameter-free optimizers.

`Schedule-Free AdamW`. [17] introduced a concept of schedule-free optimizers. Underlying idea behind his `Schedule-Free SGD` and `Schedule-Free AdamW` is to remove the scheduler with iterate averaging. Particularly, schedule-free method uses an interpolation between Polyak-Ruppert averaging [68, 73] and Primal averaging [56] for momentum update instead of usual EMA (`line 4` of Algorithm 12). To avoid undesirable behavior during scalable training the authors also propose internal warmup (see `line 7` of Algorithm 12) which uses the general number of warmup iterations parameter in the code, this gradually increases the learning rate and, at the same time, ensures `Adam`'s bias correction.

An interesting result we observe, `SF-AdamW` shows the best performance with larger number of warmup iterations compared to other methods.

Another key point — training with SF-AdamW is sensitive to the choice of beta parameters. Unlike in AdamW, these parameters serve different purposes in SF-AdamW: as β_1 acts as an interpolation parameter between two sequences, and β_2 controls the EMA of the second moment estimate, which relates to $\boldsymbol{y}$ sequence rather than the model parameters $\boldsymbol{x}$ (line 6 of Algorithm 12). For Adam it is common to analyze in theory the case, when $\beta_2 = 1 - 1/T$ [89, 10], i.e., the choice of the "optimal" beta parameters depends on the length of training. Which is also the case for SF-AdamW, making it not fully schedule-free. [28] observed this sensitivity to beta parameters, and we go beyond this ablation also.

Importantly, the authors mention that disabling the gradient norm clipping is crucial for schedule-free runs, however, we do not observe this in practice, demonstrating the contrary results.

Algorithm 12 SF-AdamW

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ, weight decay λ, β_1, β_2, warmup iterations T_{warmup}, ε.
Initialize: $\boldsymbol{z}_0 \leftarrow \boldsymbol{x}_0$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{y}_t \leftarrow (1-\beta_1)\boldsymbol{z}_t + \beta_1 \boldsymbol{x}_t$
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{y}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$
  $\gamma_t \leftarrow \gamma \sqrt{1-\beta_2^t}\min\{1, t/T_{\text{warmup}}\}$
  $\boldsymbol{z}_{t+1} \leftarrow \boldsymbol{z}_t - \gamma_t \left(\boldsymbol{g}_t/(\sqrt{\boldsymbol{v}_t} + \varepsilon) + \lambda \boldsymbol{y}_t\right)$
  $c_{t+1} \leftarrow \frac{\gamma_t^2}{\sum_{i=0}^{t}\gamma_i^2}$
  $\boldsymbol{x}_{t+1} \leftarrow (1-c_{t+1})\boldsymbol{x}_t + c_{t+1}\boldsymbol{z}_{t+1}$
end for
Return: $\boldsymbol{x}_T$

Prodigy. Improving the D-Adaptation concept [16], [54] derived an Adam-like method, which use an EMA for the learning rate (see lines 8, 9 of Algorithm 13), approximating the Adam's update of the second moment estimate EMA. The derived update reflects an EMA of $d_t \boldsymbol{g}_t$ sequence rather than $\boldsymbol{g}_t$. Idea of such a method is to come up with a scheme that is able to remove the hand-tuned learning rate via sequence which adapts during training on the fly. In Algorithm 13, d_t is such a sequence that affects botth first and second moment estimates, and evolves according to line 10.

Crucially, Prodigy does not need the learning rate tuning (typically, we initialize $\gamma = 1$), however, it still can be compatible with learning rate schedules, which we verify experimentally at scale. We also show that d_t sequence indeed acts similarly to cosine learning rate scheduler, with usually smaller learning rate at initialization and a bit larger values of it at maximum Moreover, this method scales reliably similar to AdamW, making it a promising choice for future development of parameter-free methods.

A.5 MARS

Very recently, [88] introduce MARS — a series optimizers, which incorporate modern adaptive methods [50, 9] and approximate second-order methods [24] with variance reduction update update style.

This optimization framework gave a birth to: MARS-AdamW — our main baseline which we call simply MARS, MARS-Lion and MARS-Shampoo. We mainly include MARS-AdamW in our ablation studies, but report results for other two optimizers.

The authors modified a variance reduction update introducing a sclaing parameter η, which we call variance reduction scaling in the outlined algorithms and experiments. This parameter controls the scale of gradient correction – see line 5 of Algorithms 14 to 16.

Algorithm 13 `Prodigy`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ, weight decay λ, β_1, β_2, ε.
Initialize: $d_0 \leftarrow 10^{-6}$, $\gamma \leftarrow 1$, $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$, $r_0 \leftarrow 0$, $\boldsymbol{s}_0 \leftarrow \mathbf{0}$ ▷ Optional: use scheduler on γ
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1) d_t \boldsymbol{g}_t$
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2) d_t^2 \boldsymbol{g}_t \odot \boldsymbol{g}_t$
  $\gamma_t \leftarrow \gamma \sqrt{1-\beta_2^t}/(1-\beta_1^t)$ ▷ Optional: use bias correction
  $r_t \leftarrow \sqrt{\beta_2} r_{t-1} + (1-\sqrt{\beta_2}) \gamma_t d_t^2 \langle \boldsymbol{g}_t, \boldsymbol{x}_0 - \boldsymbol{x}_t \rangle$
  $\boldsymbol{s}_t \leftarrow \sqrt{\beta_2} \boldsymbol{s}_{t-1} + (1-\sqrt{\beta_2}) \gamma_t d_t^2 \boldsymbol{g}_t$
  $d_{t+1} \leftarrow \max\{d_t, \frac{r_t}{\|\boldsymbol{s}_t\|_1}\}$
  $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \gamma_t d_t \left(\boldsymbol{m}_t / \left(\sqrt{\boldsymbol{v}_t} + d_t \varepsilon \right) + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

An important detail, we follow only the approximate scheme of `MARS`-like optimizers, i.e., we evaluate the gradient $\boldsymbol{g}_t$ in different stochasticity, meaning

$$\boldsymbol{g}_t = \nabla \mathcal{L}\left(\boldsymbol{x}_t, \boldsymbol{\xi}_t\right),$$
$$\boldsymbol{g}_{t-1} = \nabla \mathcal{L}\left(\boldsymbol{x}_{t-1}, \boldsymbol{\xi}_{t-1}\right).$$

Importantly, in the same spirit as for `SOAP` and `Muon`, the authors use `MARS`-like algorithms for layers with two and more dimensions, for 1D layers, embeds, scalar parameters and final the head layer of neural network, this method utilize `AdamW`. Such a choice allows use `MARS` in the more fast and still efficient way. Following the practices from their work, we also use `MARS` only for 2D layers.

`MARS` (`MARS-AdamW`). For `AdamW`-like algorithm, the difference occurs at the computation of $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$, where instead of the gradient, the variance reduction update $\boldsymbol{c}_t$ is used.

Algorithm 14 `MARS` (`MARS-AdamW`)

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, β_2, variance reduction scaling η, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{c}_t \leftarrow \boldsymbol{g}_t + \eta \frac{\beta_1}{1-\beta_1} \left(\boldsymbol{g}_t - \boldsymbol{g}_{t-1} \right) \|\boldsymbol{c}_t\|_2 > 1$
  $\boldsymbol{c}_t \leftarrow \boldsymbol{c}_t / \|\boldsymbol{c}_t\|_2$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1) \boldsymbol{c}_t$
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2) \boldsymbol{c}_t \odot \boldsymbol{c}_t$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1-\beta_1^t), \quad \hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1-\beta_2^t)$
  $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \gamma_t \left(\frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon} + \lambda \boldsymbol{x}_t \right)$
end for
Return: $\boldsymbol{x}_T$

`MARS-Lion`. Similarly for `Lion`-like algorithm, the authors use scaled gradient correction with the current gradient — $\boldsymbol{c}_t$.

`MARS-Shampoo`. The same holds for `MARS-Shampoo`. One key comment here, is that to compute `SVD` of the first moment estimate, the authors also use Newton-Schulz iteration [5, 25]. In our experiments we use 10 iterations of this orthogonalization scheme for `MARS-Shampoo`.

Algorithm 15 `MARS-Lion`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, variance reduction scaling η, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{c}_t \leftarrow \boldsymbol{g}_t + \eta \frac{\beta_1}{1-\beta_1} \left(\boldsymbol{g}_t - \boldsymbol{g}_{t-1}\right) \|\boldsymbol{c}_t\|_2 > 1$
  $\boldsymbol{c}_t \leftarrow \boldsymbol{c}_t / \|\boldsymbol{c}_t\|_2$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{c}_t$
  $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \gamma_t \left(\texttt{sign}\left(\boldsymbol{m}_t\right) + \lambda \boldsymbol{x}_t\right)$
end for
Return: $\boldsymbol{x}_T$

Algorithm 16 `MARS-Shampoo`

Input: Initial parameters $\boldsymbol{x}_0$, number of iterations T, learning rate γ_t, weight decay λ, β_1, variance reduction scaling η, ε.
Initialize: $\boldsymbol{m}_0 \leftarrow \mathbf{0}$, $\boldsymbol{v}_0 \leftarrow \mathbf{0}$
for $t \in [T]$ **do**
  $\boldsymbol{g}_t \leftarrow \nabla \mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\xi}_t)$
  $\boldsymbol{c}_t \leftarrow \boldsymbol{g}_t + \eta \frac{\beta_1}{1-\beta_1} \left(\boldsymbol{g}_t - \boldsymbol{g}_{t-1}\right)$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{c}_t$
  $\boldsymbol{U}_t, \boldsymbol{\Sigma}_t, \boldsymbol{V}_t \leftarrow \texttt{SVD}(\boldsymbol{m}_t)$
  $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \gamma_t \left(\boldsymbol{U}_t \boldsymbol{V}_t^\top + \lambda \boldsymbol{x}_t\right)$
end for
Return: $\boldsymbol{x}_T$

B Implementation

Our code is based on an extension of nanoGPT[5] and uses PyTorch [63] as well as FlashAttention [14]. We incorporate mixed-precision training [53], i.e., we train in `bfloat16` precision, except for normalization modules and softmax which we train in `float32`. The optimizer states are also stored in `float32`. The majority of the experiments were performed using a cluster of A100-SXM4-80GB GPUs as well as H100-HBM3-80GB. We trained both in a single GPU regime and in DDP [45] (from 2 to 8 GPUs per one run). We estimate that the full cost of all experiments for our project to roughly 30000 GPU hours.

C Model & Data

Architecture details. In our project, we use Llama-like family of models [48]. We implement the the popular in the community decoder-only transformer with SwiGLU activation functions [77], RoPE embeddings [81], RMSNorm [91]. The vocabulary is based on the GPT2 [71] tokenizer [6] and contains 50304 tokens.

The number of parameters in our models is fully configurable, and we present the exact configurations used in our experiment in Table 2.

Dataset. Our main findings are obtained on the subset of FineWeb [64] with $100\mathbf{B}$ tokens [7], cleaned and deduplicated corpus for LLM pretraining, which we split into train and validation sequences. During training, we evaluate the models with a fixed set of 32 batches of our chosen sequence length (512 for almost all experiments, the same context length as training) to establish the validation loss curves. At the end of training, we compute the full validation loss and perplexity (this loss is reported

[5] https://github.com/karpathy/nanoGPT
[6] https://github.com/openai/tiktoken
[7] https://huggingface.co/datasets/HuggingFaceFW/fineweb

Table 2: Hyperparameters for our Llama-like models.

# Parameters	124M	210M	600M
Hidden size	768	768	
# Attention heads	12	12	
# Layers	12	24	
Init std	0.02	0.02	0.02
Use bias	no	no	no
RMSNorm epsilon	0.00001	0.00001	0.00001
Positional encoding	RoPE	RoPE	RoPE

as Final Validation Loss in the figures). We also performed our initial results on the subset of OpenWebText2 dataset [21].

D Hyperparameter tuning

How do we tune hyperparameters? We perform systematic hyperparameter tuning for all algorithms, starting with smaller models (124M, 210M) and extrapolating to larger ones. Our tuning process focused on two primary settings: **Small batch setting** (32 batch size) and **Large batch setting** (256 batch size). For both settings, we use a sequence length of 512 tokens, resulting in 16k and 130k tokens per batch, respectively. If the batch cannot fit into memory, we use gradient accumulation steps, while maintaining the effective batch size.

We also include ablations on even larger batch size for 124M models, where we train on 512 batch size (260k tokens correspondingly). And larger, 583M models, we train on 3936 batch size, preserving the basic sequence length of 512, i.e., 4M tokens.

We first run multiple experiments, greed searching hyperparameters, on near Chinchilla optimal length of training using *cosine learning rate scheduler* (except for `SF-AdamW`):

- for 124M models we tune at 2.1B tokens for both small (32) and large (256) batch size setting,
- for 210M models we tune at 4.2B tokens for our large batch size setting,
- for 583M models we also consider a setting with and without z-loss.

We present the configurations for different training horizons in Tables 3 and 5.

Table 3: Lengths of training for **Small batch settings** (32×512).

# Parameters	Tokens (Iterations)						Ch. Tokens
124M	1B (64k)	2.1B (128k)	4.2B (256k)	6.3B (384k)	8.4B (512k)	16.8B (1024k)	2.5B
210M	1B (64k)	2.1B (128k)	4.2B (256k)	6.3B (384k)	8.4B (512k)	16.8B (1024k)	4.2B

Table 4: Lengths of training for **Large batch settings** (256×512).

# Parameters	Tokens (Iterations)						Chinchilla Tokens
124M	1B (8k)	2.1B (16k)	4.2B (32k)	6.3B (48k)	8.4B (64k)	16.8B (128k)	2.5B
210M	1B (8k)	2.1B (16k)	4.2B (32k)	6.3B (48k)	8.4B (64k)	16.8B (128k)	4.2B

Important to note, for larger models, we mostly kept the best hyperparameters found for the 124M model and re-tuned the learning rate and gradient clipping. We summarize this process in Appendices D.1 to D.3.

Additionally, when we report an of one particular hyperparameters, we mean that corresponding algorithm has already been tuned and, thus, we show only how one particular hyperparameter affects the overall performance.

Hyperparameters used in our WSD scheduler experiments. Once we found the best setting for each method using cosine learning rate scheduler, we are ready to obtain the optimal performance of our method with WSD scheduler [27]. Here we follow the rule of thumb from [28]:

Table 5: Lengths of training for **X-Large batch settings** (1984×512).

# Parameters	Tokens (Iterations)			Chinchilla Tokens
720M	8B (8k)	16B (16k)	48B (48k)	14.4B

- use half the optimal learning rate for the cosine scheduler,
- use 20% of iterations for cooldown phase,
- use $(1 - \sqrt{x})$ decay shape for the cooldown phase,

the only difference is that we do not employ stochastic weight averaging [29].

Therefore, we maintain most hyperparameters across optimizers, only re-tuning the learning rate. For methods like `Muon` and `MARS`, we reduce both `AdamW`'s learning rate and the learning rate for non-1D parameters. This approach ensures a fair comparison while accounting for the unique properties of each optimizer.

Indeed, this rule of thumb works better in our setting also, e.g., see the comparison between linear decay shape and $(1 - \sqrt{x})$ in

We report a series of comparisons between cosine learning rate scheduler and WSD in

Hyperparameters used in `z-loss` experiments.

D.1 124M parameters model

Table 6: `AdamW` hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.0001, \mathbf{0.0005}, 0.0008, 0.001, 0.002$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 5000, 8000$	$500, 1000, \mathbf{2000}, 3000, 8000, 32000$
Weight decay	0.1	no, $0.1, \mathbf{0.5}, 0.7$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	no, $\mathbf{0.5}, 1, 1.5$	no, $0.5, 1$
`AdamW` β_1	$0.5, \mathbf{0.8}, 0.9$	$\mathbf{0.8}, 0.9$
`AdamW` β_2	$0.95, \mathbf{0.999}$	$0.95, 0.99, \mathbf{0.999}, 0.9999$

Table 7: `ADOPT` hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	0.001	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$\mathbf{2000}, 8000, 32000$
Weight decay	0.1	no $\mathbf{0.1}, 0.5$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	no, $0.5, 1$
`ADOPT` β_1	0.9	$0.8, \mathbf{0.9}$
`ADOPT` β_2	$\mathbf{0.999}, 0.9999$	$0.5, \mathbf{0.999}, 0.9999$

Table 8: AdEMAMix hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.0001, \mathbf{0.0005}, 0.0008, 0.001, 0.002$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$\mathbf{2000}, 8000, 32000$
Weight decay	0.1	no, $0.1, \mathbf{0.5}, 0.7$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	no, $\mathbf{0.5}, 1, 1.5$	no, $\mathbf{0.5}, 1$
AdEMAMix β_1	$0.5, \mathbf{0.8}, 0.9$	$0.8, \mathbf{0.9}$
AdEMAMix β_2	0.999	$\mathbf{0.999}, 0.9999$
AdEMAMix β_3	$0.999, \mathbf{0.9999}, 0.99995$	$\mathbf{0.999}, 0.9999$
AdEMAMix α	$5, \mathbf{8}, 12$	8

Table 9: Lion hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.00005, \mathbf{0.0001}, 0.0005, 0.001$	$0.0001, 0.0005, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	3000	$\mathbf{2000}, 8000, 32000$
Weight decay	no, $0.1, 0.2, \mathbf{0.5}$	no, $0.1, \mathbf{0.5}, 0.7$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	no, $\mathbf{0.5}, 1$
Lion β_1	$0.7, \mathbf{0.9}, 0.99$	$0.5, \mathbf{0.9}$
Lion β_2	$0.9, \mathbf{0.99}, 0.999$	$\mathbf{0.99}, 0.999$

D.2 210M parameters model

D.3 600M parameters model

E Additional results

E.1 Benchmarking: 124M

In this section, we provide complete results for the benchmarking part presented in Section 4.1. We cover both the **large batch setting** and the **small batch setting**, reporting the full curves with validation loss dynamics across different training durations.

Given the quite a lot number of methods under consideration, we divide them into two groups: those that outperform AdamW and those that underperform relative to AdamW. We use AdamW loss curves as the reference point in both figures. We summarize our findings for the small batch size of 32 in Figure 12. And for the large batch size of 256 in Figures 13 and 14.

E.2 Ablations for 124M model

Fail of Sophia.

Clipping & SF-AdamW.

Betas sensitivity.

Warmup ablation. In this section we detaily describe the main part in Section 4.1.

We study the impact of batch size on the final validation loss obtained. For all methods, we sweep over warmup lengths of $\{1.56\%, 6.25\%, 25\%\}$ of the total training duration to examine each method's

Table 10: `Signum` hyperparameter tuning for our $124\mathbf{M}$ parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.0003, 0.0005, \mathbf{0.001}$	$0.0001, 0.00030.0005, 0.0003, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$2000, \mathbf{3000}$	$\mathbf{2000}, 8000, 32000$
Weight decay	no, $0, \mathbf{0.1}, 0.5$	no, $0, \mathbf{0.1}, 0.5, 0.7$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	no, $\mathbf{0.5}, 1$	no, $\mathbf{0.5}, 1$
Momentum	no, $0.9, \mathbf{0.95}$	no, $0.9, \mathbf{0.95}, 0.99$
Nesterov momentum	no, **yes**	no, **yes**

Table 11: `Muon` hyperparameter tuning for our $124\mathbf{M}$ parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate `AdamW`	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$
Learning rate `Muon`	$0.001, \mathbf{0.01}, 0.02$	$0.001, \mathbf{0.01}, 0.02$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$\mathbf{2000}, 8000, 32000$
Weight decay	no, $\mathbf{0.1}, 0.5$	no, $\mathbf{0.1}, 0.5$
Learning rate decay scheduler	**WSD**, cosine	**WSD**, cosine
Gradient clipping	no, $\mathbf{0.5}$	no, $\mathbf{0.5}, 1.0$
Momentum `Muon`	$0.9, 0.95, \mathbf{0.99}$	$\mathbf{0.95}, 0.99$
Optimizer for 1D layers	`AdamW`	`AdamW`
Optimizer for 1D layers, β_1	$0.8, 0.9$	$\mathbf{0.8}, 0.9$
Optimizer for 1D layers, β_2	$0.99, \mathbf{0.999}, 0.9999$	$0.99, 0.999, 0.9999$
Newton-Schulz a	3.4445	3.4445
Newton-Schultz b	-4.7750	-4.7750
Newton-Schultz c	2.0315	2.0315
Nesterov momentum	no, **yes**	no, **yes**

sensitivity to warmup. For `AdamW`, we extend this sweep to $\{1.56\%, 5\%, 6.25\%, 10\%, 25\%\}$. We specifically consider the 1.56% and 6.25% percentages because the former represents a typical number of warmup steps (2000) for models of our scale, while the latter (6.25% of 128000 steps) aligns with the warmup strategy used in Llama [48].

Contrary to the insights from [92], we observe that 25% of the Cinchilla optimal duration is far from being the best batch size for pretraining. We emphasize that their results were obtained for $85\mathbf{M}$ models and then extrapolated to larger scales. However, in our setting, we found the basic 2000 steps a more suitable option for warmup. 25% of Chinchilla optimal length of training, for our $124\mathbf{M}$ model is $620\mathbf{M}$ tokens.

We provide the warmup sweep for `AdamW` in Figure 19

In addition to validating the results from [92], we report the sensitivity of different optimizers to the number of warmup steps by conducting a sweep over the aforementioned percentages. A summary of this experiment — Figure 6.

`Muon` **Newton-Schulz iterations.**

Weight decay ablation.

Learning rate sensitivity. In this part of the work, we meticulously replicate the learning rate sweep process and present comprehensive results. Consistent with our experimental setup, we aim to determine the true impact of the learning rate and its transferability to longer training horizons.

Table 12: SOAP hyperparameter tuning for our $124\mathbf{M}$ parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.005, \mathbf{0.001}$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$\mathbf{2000}, 4000, 8000, 12000, 16000, 32000$
Weight decay	0.1	no, $\mathbf{0.1}, 0.5$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	no, $\mathbf{0.5}, 1$
Preconditioner dimension	10000	10000
Preconditioning frequency	$1, 5, \mathbf{10}$	$1, 5, \mathbf{10}$
SOAP β_1	$0.8, \mathbf{0.9}$	$0.8, \mathbf{0.9}, 0.95$
SOAP β_2	$0.95, 0.99, \mathbf{0.999}, 0.9999$	$0.95, 0.99, \mathbf{0.999}, 0.9999$

Table 13: Sophia hyperparameter tuning for our $124\mathbf{M}$ parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.0001, \mathbf{0.0003}, 0.0005, 0.001, 0.002$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002, 0.01$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{2000}, 3000$	$\mathbf{2000}, 8000, 32000$
Weight decay	0.1	no, $\mathbf{0.1}, 0.5$
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	no, $\mathbf{0.5}, 1$
Estimator	Gauss-Newton-Bartlett	Gauss-Newton-Bartlett
Estimator frequency	10	10
Sophia β_1	0.9	$0.8, \mathbf{0.9}$
Sophia β_2	$0.95, \mathbf{0.999}, 0.9999, 0.99999$	$0.95, \mathbf{0.999}, 0.9999, 0.99999$
Sophia ρ	$0, 0.03, \mathbf{0.04}$	$0, 0.03, \mathbf{0.04}$

For each optimizer (except Prodigy), we vary only the learning rate while maintaining the best hyperparameter settings obtained during our initial tuning (see Appendices D and D.1) on $2.1\mathbf{B}$ tokens for the $124\mathbf{M}$ parameter model. We present the results of the learning rate sweep in Figure 24.

Cosine vs WSD. We present our results for two batch size settings: 32 and 256. At first, our initial results in small batch setting on the OpenWebText2 (OWT2) dataset, we present in Figure 25.

We report the final validation loss on the FineWeb dataset for $124\mathbf{M}$ model trained on the batch size of 256. We use our tuned with cosine scheduler methods. For WSD, we follow the rule of thumb from [28]: 20% of steps for the cooldown, $1 - \sqrt{x}$ decay shape, and the learning rate is half the optimal for cosine, i.e., 0.0005 if we have the best learning rate 0.001 for the method. Additionally, we point out that we do not include stochastic weight averaging in the comparison, which might potentially enhance the performance of optimizers with WSD.

Gradient norm patterns.

E.3 Benchmarking: $210\mathbf{M}$

E.4 Ablations for $210\mathbf{M}$ model

E.5 Wall-clock performance of optimizers across models of different scale

E.6 Extension to MoEs.

Table 14: `Schedule-Free AdamW` hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.005$	$0.0001, 0.0003, 0.0005, \mathbf{0.001}, 0.002, 0.005$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$2000, 4000, \mathbf{8000}, 12000, 16000, 32000$
Weight decay	no, $0.05, \mathbf{0.1}, 0.5$	no, $0.05, \mathbf{0.1}, 0.5$
Learning rate decay scheduler	no	no
Gradient clipping	no, $\mathbf{0.5}$	no, $\mathbf{0.5}, 1$
`Schedule-Free AdamW` β_1	$\mathbf{0.9}, 0.95, 0.98$	$\mathbf{0.9}, 0.95, 0.98$
`Schedule-Free AdamW` β_2	$0.95, 0.99, 0.999, \mathbf{0.9999}, 0.99999$	$0.95, 0.99, 0.999, \mathbf{0.9999}, 0.99999$

Table 15: `Prodigy` hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate	$0.5, \mathbf{1}$	$0.5, \mathbf{1}, 2, 10, 100$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$\mathbf{3000}, 8000$	$\mathbf{2000}, 4000, 8000, 12000, 16000, 32000$
Weight decay	no, $\mathbf{0.1}, 0.5$	no, $\mathbf{0.1}, 0.5$
Learning rate decay scheduler	no, WSD, **cosine**	no, WSD, **cosine**
Gradient clipping	no, $\mathbf{0.5}, 1$	no, $\mathbf{0.5}, 1$
`Prodigy` β_1	0.9	$0.8, \mathbf{0.9}$
`Prodigy` β_2	$0.99, \mathbf{0.999}, 0.9999$	$\mathbf{0.999}, 0.9999$
`Prodigy` bias correction	no, **yes**	no, **yes**

Table 16: `MARS` (`MARS-AdamW`) hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate `AdamW`	$0.0001, 0.0005, \mathbf{0.001}, 0.003$	$0.0001, 0.0005, \mathbf{0.001}, 0.003$
Learning rate `MARS`	$0.001, \mathbf{0.003}$	$0.001, \mathbf{0.003}$
Batch size	32	256
Sequence length	512	512
Number of warmup steps	$2000, \mathbf{3000}$	$\mathbf{2000}, 8000, 32000$
Weight decay `MARS`	no, $\mathbf{0.1}$	no, $\mathbf{0.1}, 0.5$
Weight decay for 1D layers	0.1	0.1
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	0.5
Optimizer for 1D layers	`AdamW`	`AdamW`
Optimizer for 1D layers β_1	$\mathbf{0.8}, 0.9$	$\mathbf{0.8}, 0.9, 0.95$
Optimizer for 1D layers β_2	$0.95, 0.99, \mathbf{0.999}$	$0.95, 0.99, \mathbf{0.999}$
`MARS` β_1	$0.9, \mathbf{0.95}$	$0.9, \mathbf{0.95}$
`MARS` β_2	$0.95, \mathbf{0.99}$	$0.95, \mathbf{0.99}$
VR scaling factor η	$0.024, \mathbf{0.025}$	$0.024, \mathbf{0.025}$

Table 17: MARS-Lion hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate Lion	**0.0001**, 0.0005, 0.001, 0.003	**0.0001**, 0.0005, 0.001, 0.003
Learning rate MARS	**0.0001**, 0.001, 0.003	**0.0001**, 0.001, 0.003
Batch size	32	256
Sequence length	512	512
Number of warmup steps	2000, **3000**	**2000**, 8000, 32000
Weight decay MARS	no, **0.1**	no, **0.1**, 0.5
Weight decay for 1D layers	0.1	0.1
Learning rate decay scheduler	**WSD**, cosine	**WSD**, cosine
Gradient clipping	0.5	0.5
Optimizer for 1D layers	Lion	Lion
Optimizer for 1D layers β_1	0.8, **0.9**	0.8, **0.9**, 0.95
Optimizer for 1D layers β_2	0.95, 0.99, **0.999**	0.95, 0.99, **0.999**
MARS β_1	0.9, **0.95**	0.9, **0.95**
MARS β_2	0.95, **0.99**	0.95, **0.99**
VR scaling factor η	0.024, **0.025**	0.024, **0.025**

Table 18: MARS-Shampoo hyperparameter tuning for our 124M parameter large language models. Bold hyperparameters are the best.

Hyperparameter	Small batch setting	Large batch setting
Learning rate Shampoo	0.0001, 0.0005, **0.001**, 0.003	0.0001, 0.0005, **0.001**, 0.003
Learning rate MARS	0.001, **0.003**	0.001, **0.003**
Batch size	32	256
Sequence length	512	512
Number of warmup steps	2000, **3000**	**2000**, 8000, 32000
Weight decay MARS	no, **0.1**	no, **0.1**, 0.5
Weight decay for 1D layers	0.1	0.1
Learning rate decay scheduler	WSD, **cosine**	WSD, **cosine**
Gradient clipping	0.5	0.5
Optimizer for 1D layers	Shampoo	Shampoo
Optimizer for 1D layers β_1	0.8, **0.9**	0.8, **0.9**, 0.95
Optimizer for 1D layers β_2	0.95, 0.99, **0.999**	0.95, 0.99, **0.999**
MARS β_1	0.9, **0.95**	0.9, **0.95**
MARS β_2	0.95, **0.99**	0.95, **0.99**
VR scaling factor η	0.024, **0.025**	0.024, **0.025**

Table 19: Hyperparameters for our Llama-like models for the wall-clock experiments.

# Parameters	30M	52M	80M	124M	150M	210M	360M	720M	1B
Hidden size	384	512	768	768	768	768	1024	2048	1792
# Attention heads	6	8	6	12	12	12	16	16	14
# Layers	8	8	6	12	16	24	24	12	24
Init std	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
Use bias	no	no	no	no	no	no	no	no	no
RMSNorm epsilon	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001	0.00001
Positional encoding	RoPE	RoPE	RoPE	RoPE	RoPE	RoPE	RoPE	RoPE	RoPE

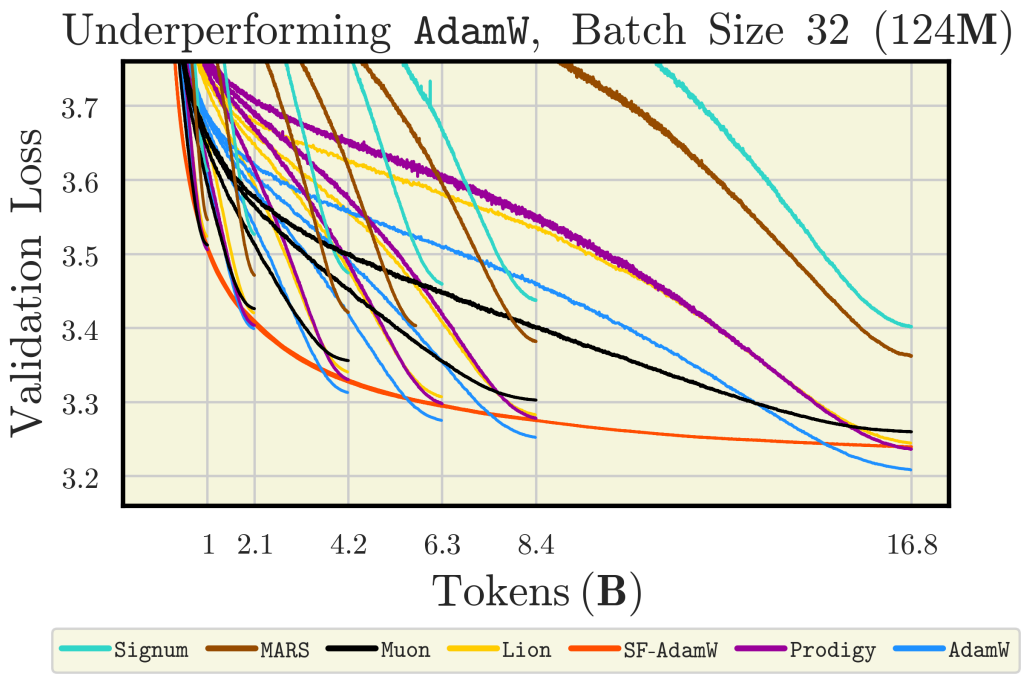


Figure 11: **Ranking of optimizers in the small-batch setting.** In the small-batch setting AdamW outperforms most of the optimizaers we study.

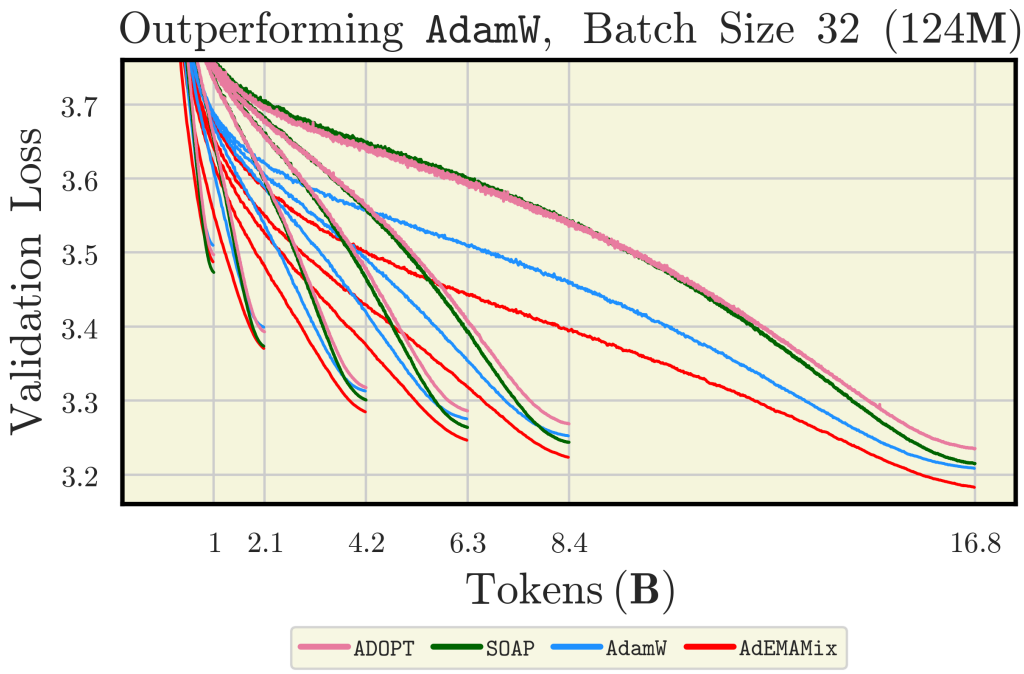


Figure 12: **Ranking of optimizers in the small-batch setting.** Here only AdEMAMix, SOAP and ADOPT show a remarkable performance compared to AdamW.

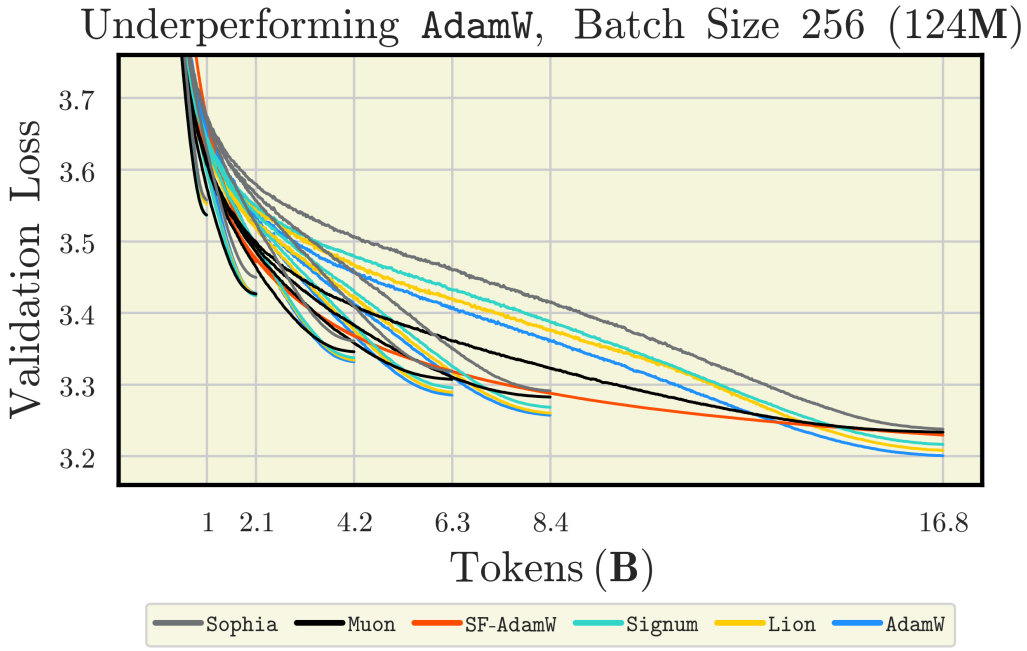


Figure 13: **Ranking of optimizers in the large-batch setting.**

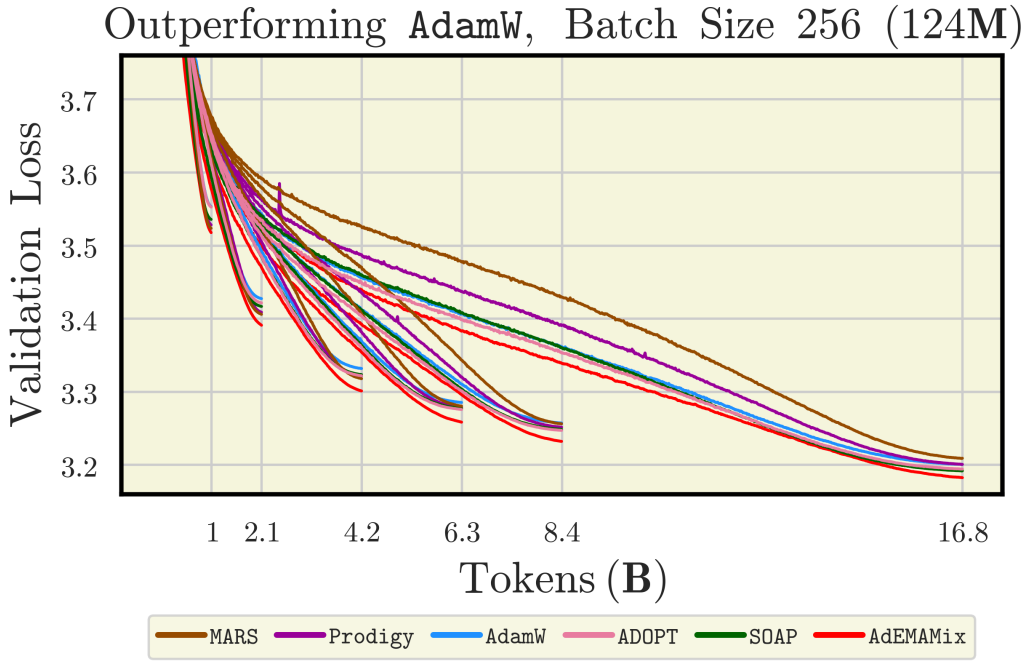
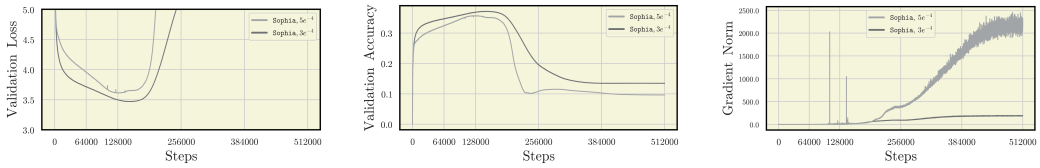
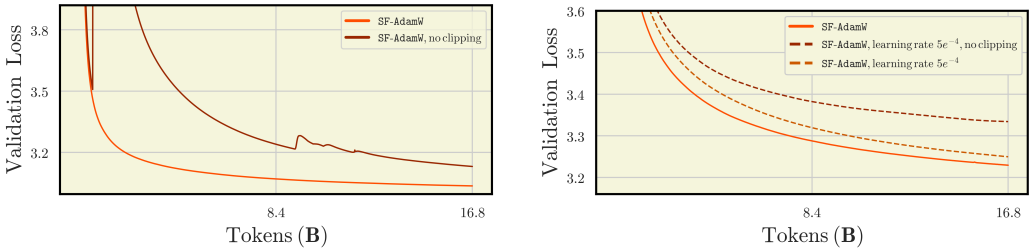


Figure 14: **Ranking of optimizers in the large-batch setting.**



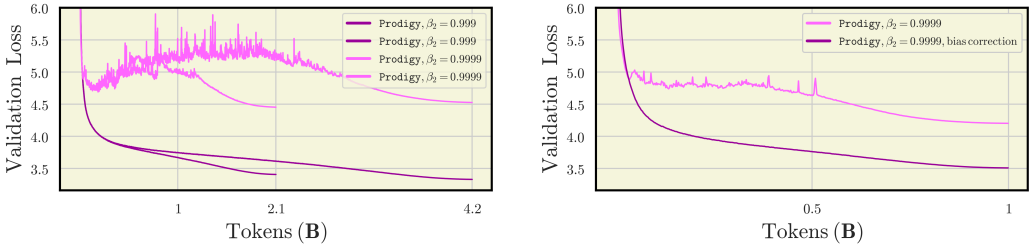
(a) (b) (c)

Figure 15: `Sophia` **diverges in the small-batch setting even with sufficiently small learning rate.**



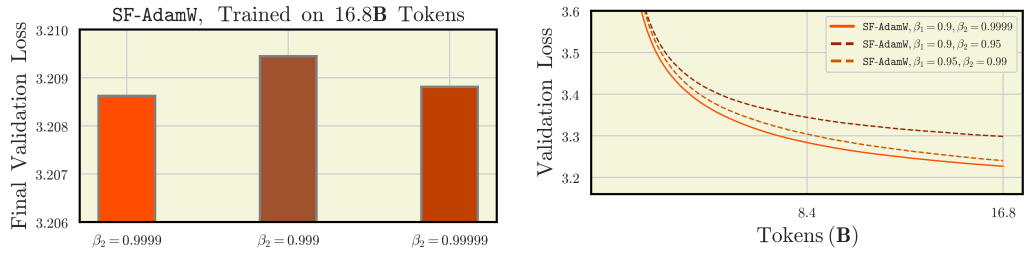
(a) (b)

Figure 16: **Clipping is significant for** `Schedule-Free`.



(a) (b)

Figure 17: `Prodigy` **is sensitive to beta parameters in the small-batch setting**.



(a) (b)

Figure 18: **Impact of beta parameters on** `Schedule-Free`.

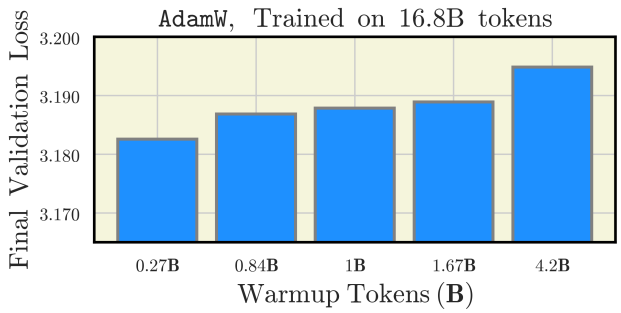


Figure 19: **Warmup sweep for `AdamW`.** We observe that the smaller yet reasonable warmup value is the best, however, this is not true for other methods like `Signum` and `SF-AdamW` (see **??**).

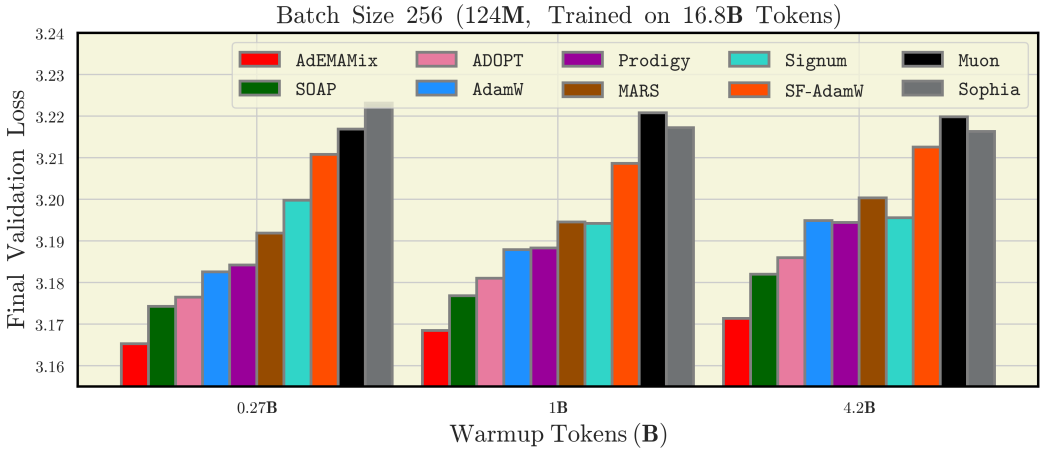


Figure 20: **Warmup ablation.** We report the final validation loss on the FineWeb dataset for 124M model trained on the batch size of 256. We sweep over the batch sizes of $\{1.56\%, 6.25\%, 25\%\}$ of the length of training, which corresponds to $\{2000, 8000, 32000\}$k iterations, respectively.

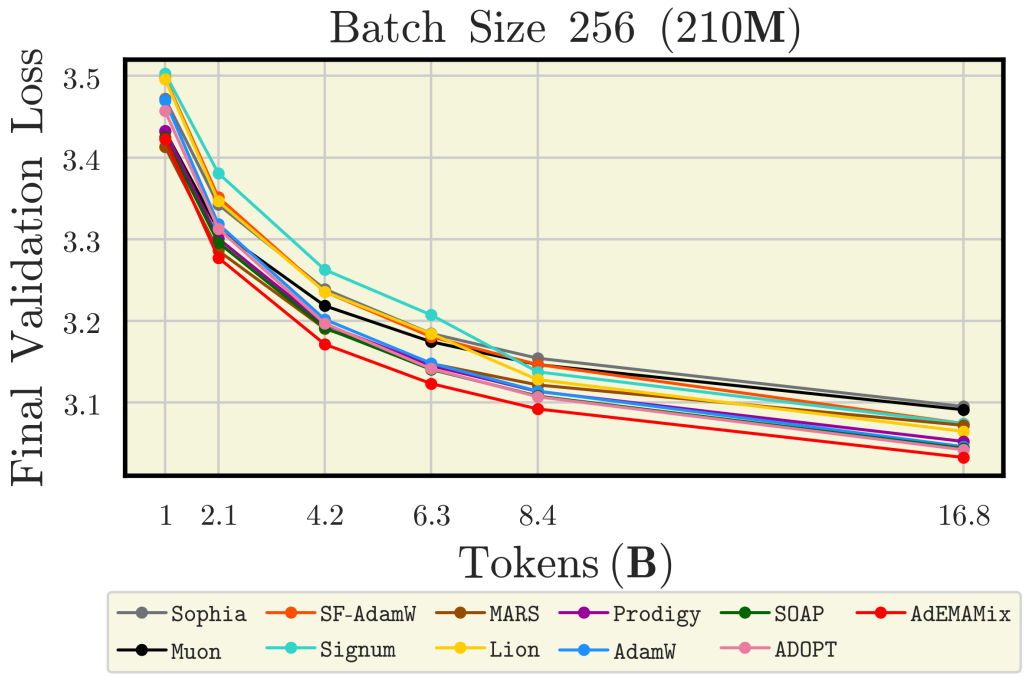


Figure 21: **Ranking of optimizers in the large-batch setting.**

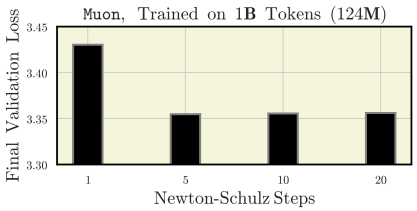


Figure 22: `Muon`**'s dependence on the number of Newton-Schulz iterations.**

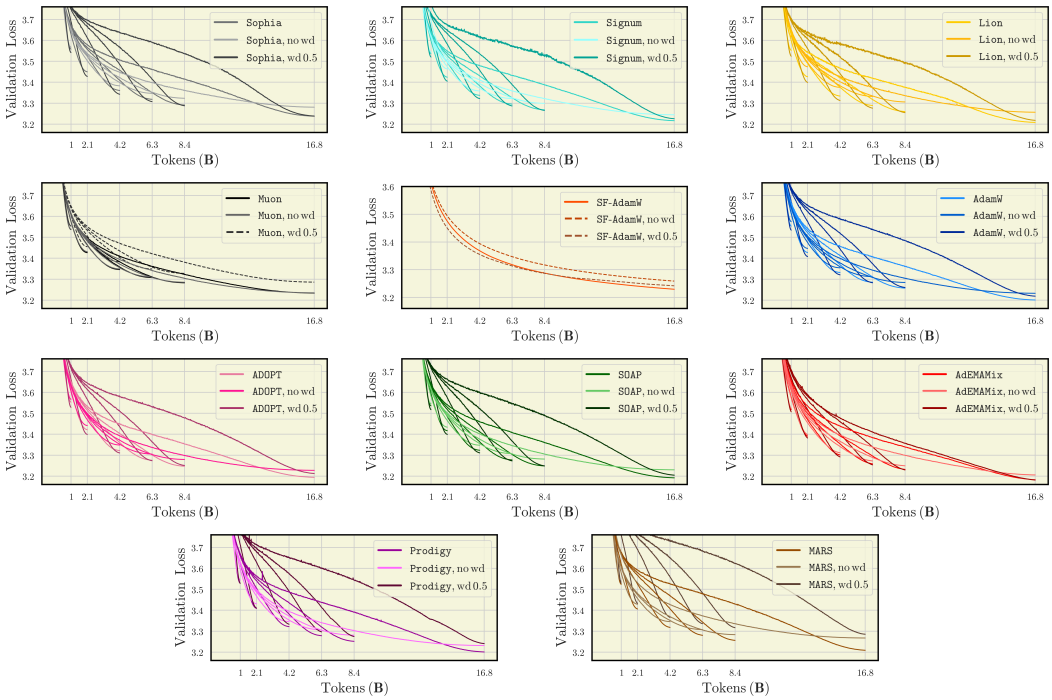


Figure 23: **Larger weight decay achieves significantly better results when training on fewer tokens.** We report the final validation loss on the FineWeb dataset for 124M model trained on the batch size of 256. We observe that the mmajority of runs with the large weight decay of 0.5 consistently outperform the same optimizer with weight decay of 0.1 for all training durations except for the last one. Notably, `Signum` and `Lion` with large weight decay perform even better than `AdamW` with the same learning rate. We also consider a setting without weight decay. We observe that this is suboptimal for most of other optimizers, while the typical weight decay of 0.1 remains the best for large training durations. An interesting thing we observe for optimizers that train one dimensional and two dimensional parameters in a different way — `Muon`, `MARS`. Indeed, the corresponding runs with the weight decay of 0.5 are always worse than then 0.1 baseline and, in some cases, even worse than runs without weight decay. For `Muon`, we connect this effect to its algorithmic design, where weight decay is not used to optimize matrix parameters (see Algorithm 7). For `MARS`, we only vary the weight decay that corresponds to matrix parameters, while keeping 0.1 for all scalar, one dimensional and final layer parameters. In this case, we conclude that the gap between large and small weight decay values narrows significantly faster.

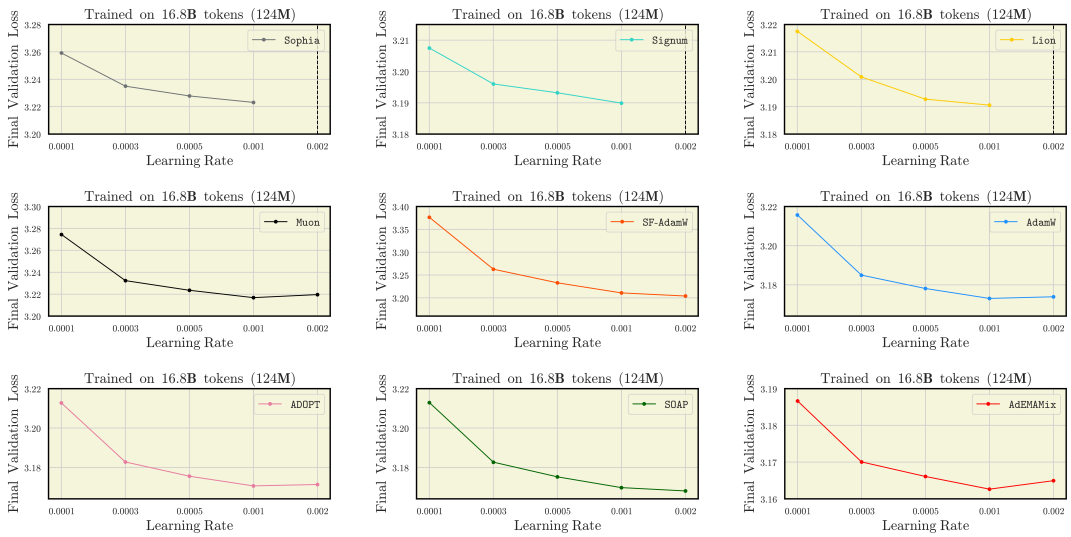


Figure 24: **Learning rate sensitivity.** We report the final validation loss on the FineWeb dataset for 124M model trained on the batch size of 256. In the current setting, only `SOAP` and `SF-AdamW` reach the better performance with the large leraning rate of 0.02. On the other hand, `Sophia` and all sign-based methods (`Signum` and `Lion`) diverge with this value of the learning rate.

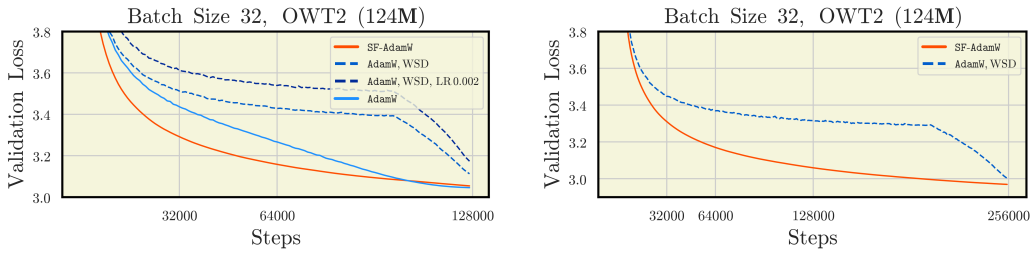


Figure 25: **WSD scheduler underperforms both `AdamW` with cosine scheduler and `SF-AdamW`.** Once the learning rate and beta parameters of `SF-AdamW` and `AdamW` are properly tuned, we observe a surprisingly large gap in performance between WSD scheduler and its competitors. Figure **(b)** suggests that this gap may potentially diminish with extended training. To investigate this further, we conduct a scalable comparison between tuned WSD and cosine baselines across longer training horizons.

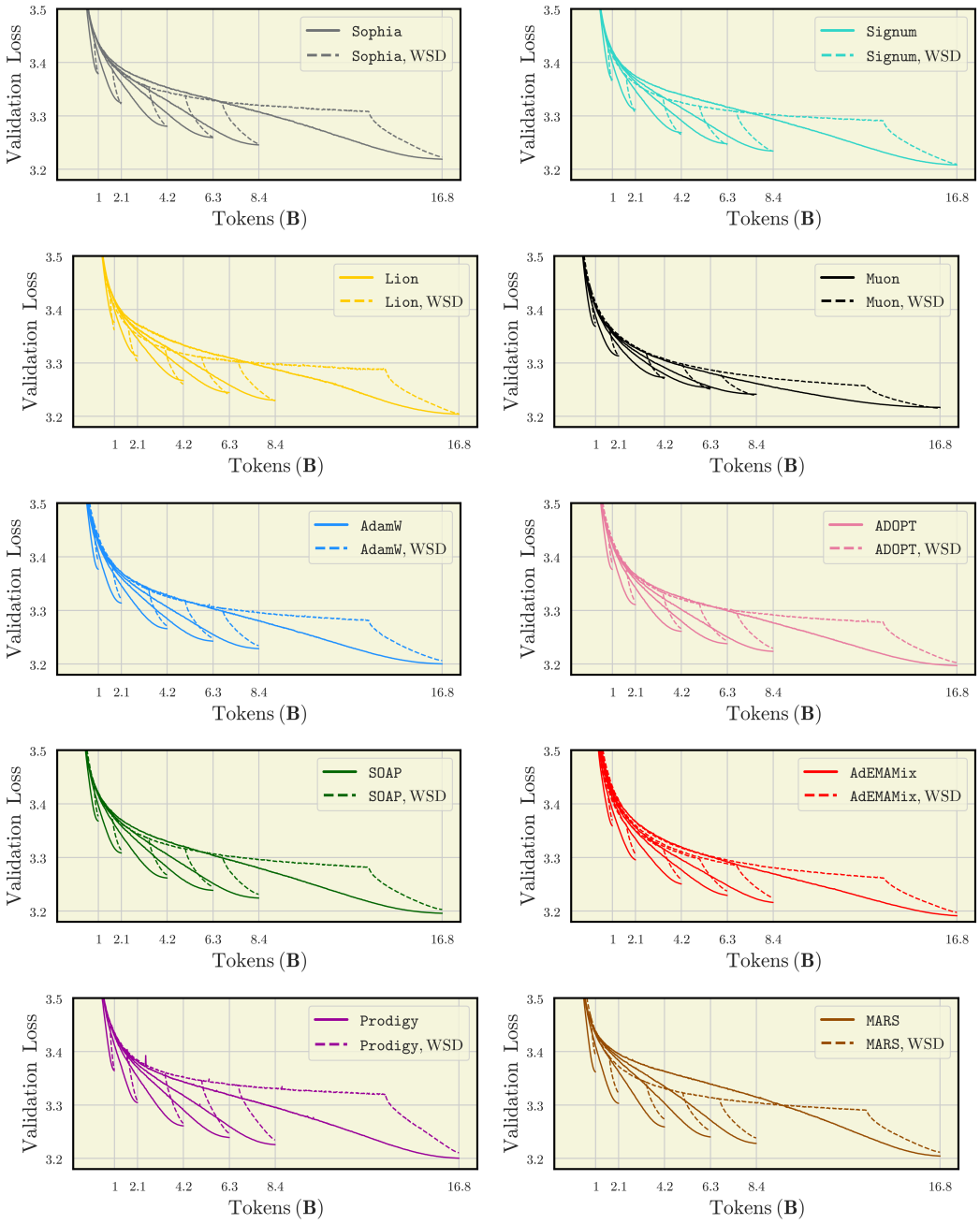


Figure 26: **WSD scheduler underperforms cosine.** We report the final validation loss on the FineWeb dataset for 124M model trained on the batch size of 256. We observe that WSD still can match the performance of cosine on `Sophia`, `Signum`, `Lion`, i.e., on the sign-based methods, and even outperform for `Muon`. Although the gap in performance is not particularly significant, but for benchmarking purposes, we decide to stick to the cosine scheduler because those gap still plays a substantial role in our setup.

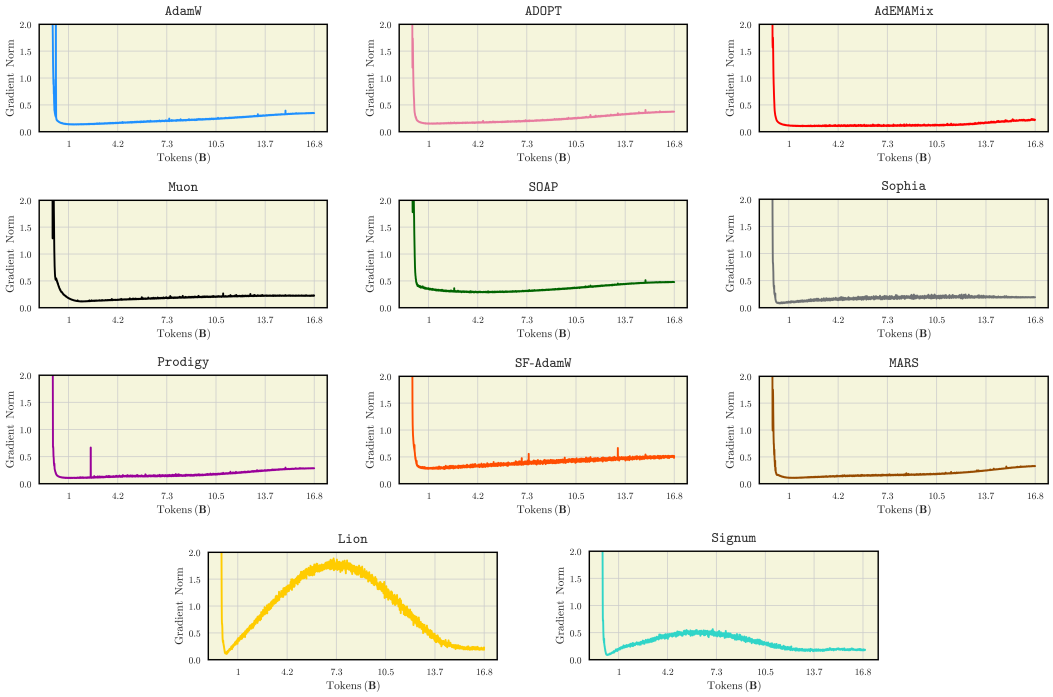


Figure 27: **Gradient Norm patterns for cosine scheduler.**

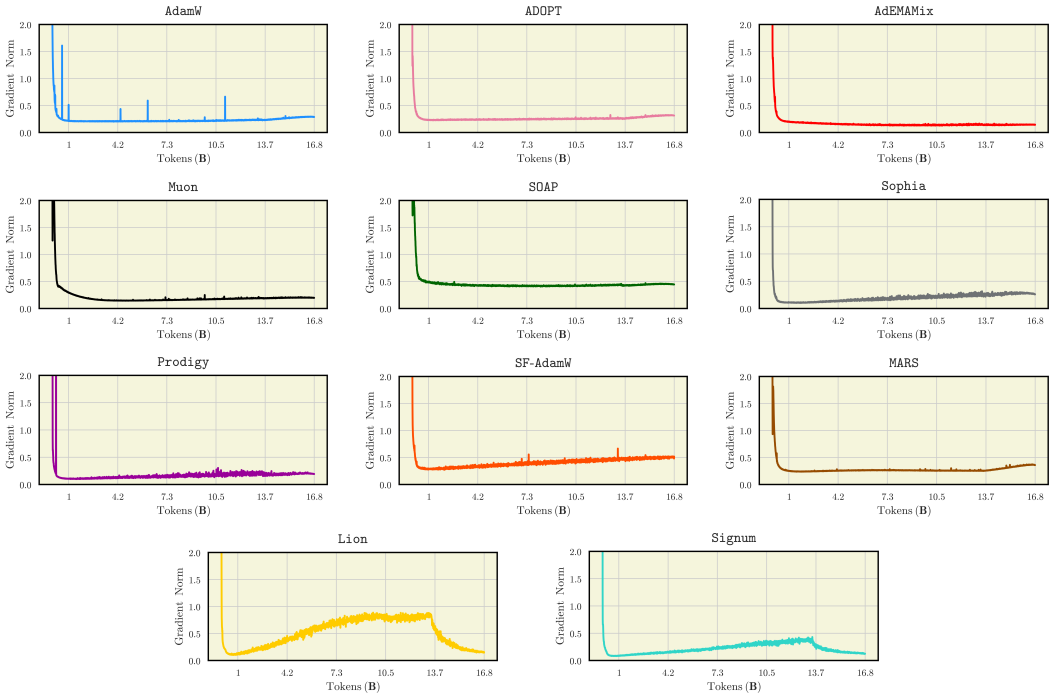


Figure 28: **Gradient Norm patterns for WSD scheduler.**

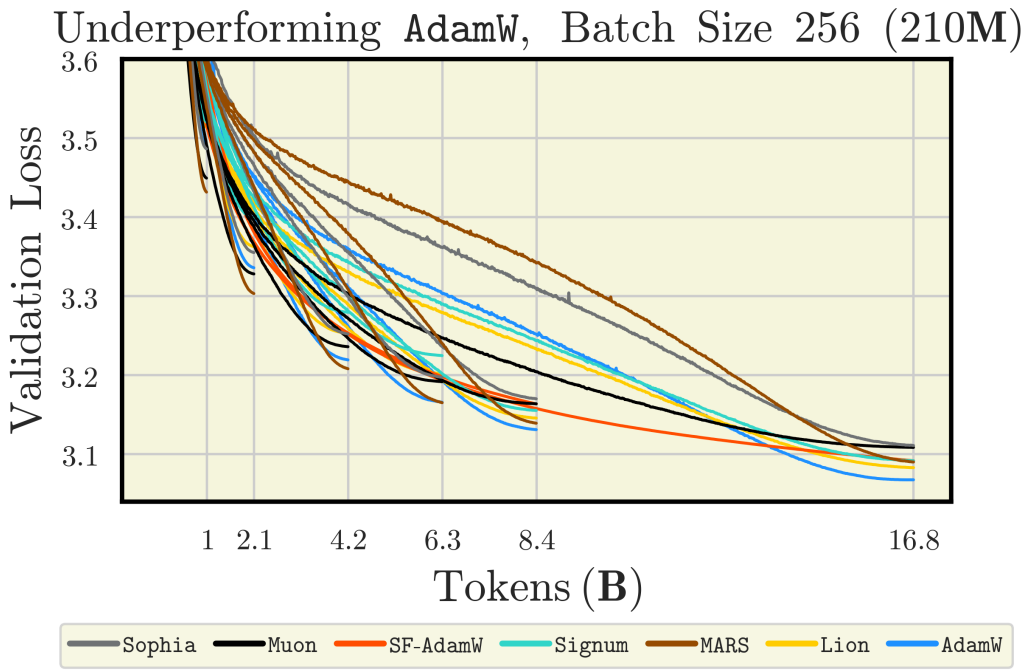


Figure 29: **Ranking of optimizers in the large-batch setting.**

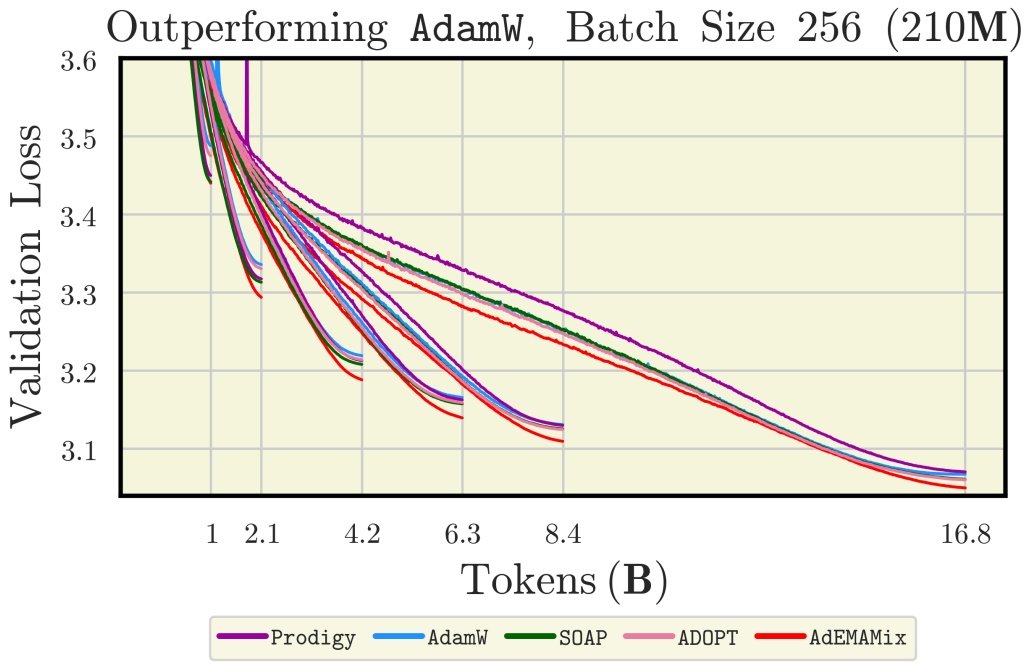


Figure 30: **Ranking of optimizers in the large-batch setting.**