OPT2024: 16th Annual Workshop on Optimization for Machine Learning

SOAP: Improving and Stabilizing Shampoo using Adam

Nikhil Vyas[*] NIKHIL@G.HARVARD.EDU
Harvard University

Depen Morwani[*] DMORWANI@G.HARVARD.EDU
Harvard University

Rosie Zhao[†] ROSIEZHAO@G.HARVARD.EDU
Harvard University

Itai Shapira[†] ITAISHAPIRA@G.HARVARD.EDU
Harvard University

David Brandfonbrener DAVID_BRANDFONBRENER@G.HARVARD.EDU
Kempner Institute at Harvard University

Lucas Janson LJANSON@FAS.HARVARD.EDU
Harvard University

Sham Kakade SHAM@SEAS.HARVARD.EDU
Kempner Institute at Harvard University

Abstract

There is growing evidence of the effectiveness of Shampoo, a higher-order preconditioning method, over Adam in deep learning optimization tasks. However, Shampoo's drawbacks include additional hyperparameters and computational overhead when compared to Adam, which only updates running averages of first- and second-moment quantities. This work establishes a formal connection between Shampoo (implemented with the 1/2 power) and Adafactor — a memory-efficient approximation of Adam — showing that Shampoo is equivalent to running Adafactor in the eigenbasis of Shampoo's preconditioner. This insight leads to the design of a simpler and computationally efficient algorithm: **S**hampo**O** with **A**dam in the **P**reconditioner's eigenbasis (SOAP).

With regards to improving Shampoo's computational efficiency, the most straightforward approach would be to simply compute Shampoo's eigendecomposition less frequently. Unfortunately, as our empirical results show, this leads to performance degradation that worsens with this frequency. SOAP mitigates this degradation by continually updating the running average of the second moment, just as Adam does, but in the current (slowly changing) coordinate basis. Furthermore, since SOAP is equivalent to running Adam in a rotated space, it introduces only one additional hyperparameter (the preconditioning frequency) compared to Adam. We empirically evaluate SOAP on language model pre-training with 360m and 660m sized models. In the large batch regime, SOAP reduces the number of iterations by over 40% and wall clock time by over 35% compared to AdamW, with approximately 20% improvements in both metrics compared to Shampoo.

1. Introduction

With ever-increasing costs of LLM training, optimization efficiency has become a central question in the field of deep learning. Several recent works have tackled this challenge by addressing both the memory

[*] Equal contribution.
[†] Equal contribution.

© N. Vyas, D. Morwani, R. Zhao, I. Shapira, D. Brandfonbrener, L. Janson & S. Kakade.

[46, 50] and compute [1] footprint of optimizers. In Algoperf [4], a recent optimization efficiency benchmark, Shampoo [17], a second-order algorithm, outperformed all other submissions, including Adam [22], reducing wall-clock time by 28%. Higher-order preconditioning has also been applied in large-scale training runs, such as Gemini-1.5 Flash [14].

The success of Shampoo has drawn increasing attention from the deep learning community. Several works have explored ways to scale Shampoo by improving its memory and compute efficiency [1, 43, 46]. Other research [32] has examined the theoretical foundations of Shampoo and proposed minor adjustments (such as using power $1/2$ rather than $1/4$) that align with prior empirical findings [1]. Moreover, Morwani et al. [32] also showed that Shampoo with the aforementioned modifications is close to the optimal Kronecker approximation of the Adagrad [10] optimizer.

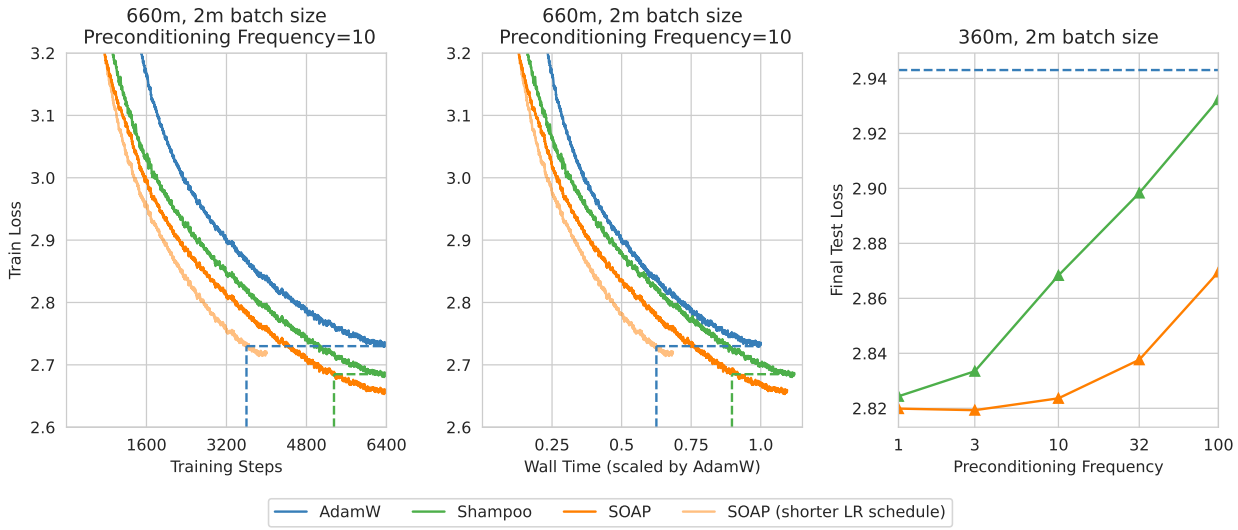

Figure 1: Comparing performance of tuned runs for AdamW, Shampoo (using DistributedShampoo [43] implementation) and SOAP. In left and middle figures, Shampoo and SOAP use a preconditioning frequency of 10. The "shorter LR schedule" plot is where we tuned the cosine decay so as to achieve the same terminal performance as AdamW. There we observe a $\geq 40\%$ reduction in the number of iterations and a $\geq 35\%$ reduction in wall clock time compared to AdamW, and approximately a 20% reduction in both metrics compared to Shampoo. In the right figure we ablate preconditioning frequency and observe a slower degradation of performance of SOAP as compared to Shampoo. See Section 5 for a discussion of experimental results and ablation of batch size and Section 4 for experimental methodology.

Our first contribution is demonstrating that the variant of Shampoo proposed by Morwani et al. [32] is equivalent[1] to running Adafactor [42, 48] in the eigenbasis provided by Shampoo's preconditioner. This interpretation of Shampoo connects it to a broader family of methods (e.g. [15]) that design second-order algorithms by running a first-order method in the eigenbasis provided by a second-order method. Building on this insight, we can explore a broader design space for combining first and second order methods. Many of our design choices are a synthesis of conceptual ideas from prior works of Anil et al. [1], George et al. [15], Morwani et al. [32] as well as implementation ideas from works of Wang et al. [46], Zhao et al. [50].

1. Given this connection, the results of Morwani et al. [32] can be interpreted as showing that the eigenbasis provided by Shampoo's preconditioner is close to the "optimal" basis for running Adafactor.

Explicitly, we study SOAP (**S**hampo**O** with **A**dam in the **P**reconditioner's eigenbasis) an algorithm that runs AdamW in the eigenbasis provided by Shampoo. Our main contributions are as follows:

- We make a formal connection between the Shampoo and the Adafactor algorithm. This insight leads us to consider the SOAP algorithm, which runs AdamW in the preconditioned space provided by Shampoo.
- SOAP outperforms both Shampoo and Adam in LLM pre-training tasks with model sizes 360m and 660m, even after extensive hyperparameter tuning of Shampoo.
- SOAP reduces the number of hyperparameters compared to Shampoo, resulting in only one additional hyperparameter compared to AdamW: preconditioning frequency. Further, SOAP demonstrates greater robustness to large preconditioning frequency compared to Shampoo.

We should also note that while similar algorithmic variants have been discussed in the literature (e.g. see the appendix of Anil et al. [1]), we are the first to systematically evaluate it.

Organization: In Section 3, we start by showing an equivalence between Shampoo (with exponent 1/2) and running Adafactor in the eigenspace given by Shampoo, then with this equivalence as the starting point we describe SOAP. In Section 4, we provide our experimental methodology and in Section 5, we compare the performance of AdamW, Shampoo and SOAP on language modeling tasks. In Appendices F.2 and F.3 we discuss the the space and time complexity of SOAP and how it can be improved.

2. Notation and Background

We denote the weight matrix of a neural network layer by $W \in \mathbb{R}^{m\times n}$, and the corresponding gradient by $G \in \mathbb{R}^{m\times n}$. At a given time step t, these are denoted as W_t and G_t, respectively. For a batch of inputs at time t, denoted by B_t, the loss and its gradient evaluated at W_t are represented as $\phi_{B_t}(W_t)$ and $\nabla_W \phi_{B_t}(W_t)$, respectively.

Adagrad [10] is an online learning second-order algorithm that maintains a preconditioner $H \in \mathbb{R}^{mn\times mn}$. If the vectorized gradient at time t is denoted by g_t (i.e., $g_t = \mathrm{vec}(G_t) \in \mathbb{R}^{mn}$), then the update of the preconditioner and the vectorized weights $w_t \in \mathbb{R}^{mn}$ with learning rate η is given by

$$H_t = H_{t-1} + g_t g_t^\top; \quad w_t = w_{t-1} - \eta H_t^{-1/2} g_t$$

Adam [22], a widely used first-order optimization algorithm in deep learning is a diagonal approximation of Adagrad. It maintains an exponential moving average of the gradients G_t (denoted as M_t) and of element-wise squared gradients G_t^2 (denoted as V_t) for a given weight matrix W. Its update rule with learning rate η is given by

$$W_t \leftarrow W_{t-1} - \eta \frac{M_t}{\sqrt{V_t}},$$

where the division is performed element-wise.

Adafactor [42, 48], a variant of Adam, replaces V_t with its best rank-1 approximation V_t' to reduce memory usage. While the original Adafactor paper [42] proposed additional modifications, such as changes to the learning rate schedule, we focus on the version of Adafactor proposed in recent works [48, 52], whose update with learning rate η is given by

$$W_t \leftarrow W_{t-1} - \eta \frac{M_t}{\sqrt{V_t'}}.$$

Shampoo [18] is a second-order optimization algorithm that approximates Adagrad and maintains two preconditioners, $L_t \in \mathbb{R}^{m\times m}$ and $R_t \in \mathbb{R}^{n\times n}$, for a given weight matrix $W \in \mathbb{R}^{m\times n}$. The updates for the preconditioners and the weights with learning rate η are as follows:

$$L_t \leftarrow L_{t-1} + G_t G_t^T; \quad R_t \leftarrow R_{t-1} + G_t^T G_t; \quad W_t \leftarrow W_{t-1} - \eta L_t^{-1/4} G_t R_t^{-1/4}.$$

In practice, Shampoo is implemented with several other modifications such as layerwise learning rate grafting and exponents other than $-1/4$. We will use the DistributedShampoo [43] implementation which has these variations available as hyperparameters.

3. Algorithm

3.1. Theory

We begin by describing an equivalence between Shampoo and running Adafactor in the eigenbasis of the Shampoo preconditioner. For simplicity we omit momentum but the equivalence also holds with momentum. For this equivalence we use Shampoo with the following modifications from the original Shampoo optimizer [18]:

1. We use power $1/2$ instead of power $1/4$. This was already recommended in practical implementations [1, 43] and a theoretical connection between optimal Kronecker approximation of Adagrad [10] preconditioner and Shampoo with power $1/2$ was established in [32].

2. We also use the scalar correction to per layer learning rates described in Morwani et al. [32], Ren and Goldfarb [41].

3. Instead of the running average of L and R across time steps, we use dataset averages.

With these changes in place, we define two algorithms which are "idealized" forms of Shampoo with power $1/2$ and running Adafactor in Shampoo's eigenspace. We formally show in Appendix B that these two algorithms are *in fact equivalent*.

Claim 1 (Informal, see Appendix B) *Running "idealized" form of Shampoo with power $1/2$ and running "idealized" Adafactor in Shampoo's eigenspace (see Algorithms 2 and 3 respectively in the Appendix) are equivalent.*

While these two algorithms are equivalent in their idealized forms, practical considerations reveal some differences. Firstly, the algorithms differ when using running averages instead of dataset averages. Secondly, and more significantly in practice, we do not invert or compute the eigenvector decomposition of L and R at every step. This means that the "adaptivity" of learning rates in Shampoo is limited[2] to the updates of L and R. In contrast, with Adafactor in Shampoo's eigenspace, the second moment estimates can be updated at every step as they are computationally inexpensive. Additionally, instead of using Adafactor, we can opt[3] for Adam, which offers more generality. Combining these insights leads to Algorithm 1 which can be interpreted as running Adam in Shampoo's eigenspace. Further implementation details of our algorithm are in Appendix C.

2. We note that practical implementations of Shampoo use grafting which allows for learning rate adaptivity at every step, but this adaptivity is restricted to a single scalar per layer.
3. Though using AdamW over Adafactor only gives very small improvements in performance, see Figure 5 and Appendix F.2. We also note that one can use any other diagonal preconditioner based optimizer in place of Adam, such as Lion [3], Sophia [26] or Schedule-Free AdamW [5].

Algorithm 1 Single step of SOAP for a $m \times n$ layer. Per layer, we maintain four matrices: $L \in \mathbb{R}^{m \times m}$, $R \in \mathbb{R}^{n \times n}$ and $V, M \in \mathbb{R}^{m \times n}$. For simplicity we ignore the initialization and other boundary effects such as bias correction. Hyperparameters: Learning rate η, betas $= (\beta_1, \beta_2)$, epsilon ϵ, and preconditioning frequency f.
An implementation of SOAP is available at https://github.com/nikhilvyas/SOAP.

Sample batch B_t.
$G \in \mathbb{R}^{m \times n} \leftarrow -\nabla_W \phi_{B_t}(W_t)$
$G' \leftarrow Q_L^T G Q_R$
$M \leftarrow \beta_1 M + (1 - \beta_1) G$
$M' \leftarrow Q_L^T M Q_R$
{Now we "run" Adam on G'}
$V \leftarrow \beta_2 V + (1 - \beta_2)(G' \odot G')$ {Elementwise multiplication}
$N' \leftarrow \frac{M'}{\sqrt{\hat{V}} + \epsilon}$ {Elementwise division and square root}
{Now that we have preconditioned by Adam in the rotated space, we go back to the original space.}
$N \leftarrow Q_L N' Q_R^T$
$W \leftarrow W - \eta N$
{End of gradient step, we now update L and R and possibly also Q_L and Q_R. }
$L \leftarrow \beta_2 L + (1 - \beta_2) G G^T$
$R \leftarrow \beta_2 R + (1 - \beta_2) G^T G$
if t % f == 0 **then**
    $Q_L \leftarrow \texttt{Eigenvectors}(L, Q_L)$
    $Q_R \leftarrow \texttt{Eigenvectors}(R, Q_R)$
end if

The main focus of the next sections will be to explore the empirical performance of this algorithm and its variations.

4. Experimental Methodology

Hyperparameter tuning: We begin with hyperparameter values suggested by prior research for both AdamW and Distributed Shampoo (e.g., $\beta_2 = 0.95$). Initially, we conduct a learning rate sweep to determine the optimal learning rate for each optimizer. Once the optimal learning rate is identified, we perform two-dimensional sweeps for each of the remaining hyperparameters, where we vary the selected hyperparameter alongside the learning rate. The purpose of these sweeps is to demonstrate that our default hyperparameter settings are near-optimal, disregarding potential interactions between two non-learning-rate hyperparameters. A detailed discussion of the hyperparameter sweeps is provided in Appendix D.

Throughput Measurement: We evaluate the throughput of each optimizer by measuring the number of tokens processed per second. At present, we perform these measurements on a single H100 GPU and utilize gradient accumulation to accommodate large batch sizes. While this approach may seem to disadvantage AdamW— as the overhead of Shampoo/SOAP is compared against multiple gradient accumulation steps— it is important to note that the overhead of Shampoo/SOAP can be amortized across layers by distributing the updates across multiple GPUs. This technique is employed in the distributed implementation of Shampoo [43]. A comprehensive comparison of distributed implementations of these algorithms is left to future work.

5. Language Modeling Experiments

In this section we focus on empirically comparing AdamW, DistributedShampoo, and SOAP on language modeling tasks. Within the main body we focus on highlighting the efficiency benefits and the robustness of SOAP to preconditioning frequency; in Appendix E.2, we present additional experiments showing the impact of decreasing the batch size on the performance of Shampoo and SOAP— even in this setting, we find SOAP consistently outperforms both Shampoo and AdamW.

5.1. Measuring Efficiency Benefits

In Figure 1 (left and middle) we show train loss curves of 360m and 660m models with 2m token batch size for AdamW, Shampoo, and SOAP, where SOAP outperforms the other two. To directly calculate the efficiency benefit of SOAP, we also run SOAP with cosine decay for a shorter lr schedule, as shown in Figure 1. This allows us to approximate the following efficiency benefits (when setting batch size to 2m and preconditioning frequency to 10): $\geq 40\%$ reduction in number of iterations and $\geq 35\%$ reduction in wall clock time as compared to AdamW; $\approx 20\%$ reduction in iterations and wall clock time as compared to Shampoo. We include analogous figures for 360m models in Appendix E.1.

Simply running SOAP for the same duration as Shampoo and AdamW cannot be directly used to calculate the efficiency benefit (in terms of training steps or wall-clock time) of using SOAP since we use a cosine schedule. Therefore, we run SOAP on $.5, .625, .75$ and $.875$ fraction of the training data and fit a scaling law of the form $a + bN^{-\beta}$ through the final losses obtained, where N represents the number of training points and a, b, β are the parameters of the fit. We show these points and the corresponding scaling laws obtained in Figure 2. This scaling law is then used to calculate the efficiency benefit in terms of training steps and wallclock time as shown in Figure 2 in Appendix E.1.

5.2. Effect of Frequency of Finding Eigenvectors/Inverse

In Figure 1 (right), we compare SOAP and Shampoo with respect to preconditioning frequency. We observe the following:

- For all frequencies we tried from 1 to 100, both optimizers outperform AdamW.
- At frequency 1, SOAP and Shampoo are quite close in performance.
- At higher frequencies, the performance of both SOAP and Shampoo degrades but SOAP's performance degrades significantly slower than Shampoo's.

6. Discussion and Future Work

We study an optimizer called SOAP: **S**hampo**O** with **A**dam in the **P**reconditioner's eigenbasis. We show that SOAP outperforms both AdamW and Shampoo in language modeling tasks and show that it is more robust to changes in preconditioning frequency than Shampoo. For future work, we would like to explore further improvements to the design of SOAP, in particular, related to using lower precision for the preconditioners as well as a better distributed implementation. We would also like to explore the performance of SOAP on other domains such as vision.

References

[1] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

[2] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SkkTMpjex.

[3] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/9a39b4925e35cf447ccba8757137d84f-Abstract-Conference.html.

[4] 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.

[5] Aaron Defazio, Xingyu Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. The road less scheduled. *CoRR*, abs/2405.15682, 2024. doi: 10.48550/ARXIV.2405.15682. URL https://doi.org/10.48550/arXiv.2405.15682.

[6] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pages 7480–7512. PMLR, 2023.

[7] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=shpkpVXzo3h.

[8] Documentation. torch.linalg.eigh documentation. https://web.archive.org/web/20240519213242/https://pytorch.org/docs/stable/generated/torch.linalg.eigh.html, 2024.

[9] 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. URL http://jmlr.org/papers/v12/duchi11a.html.

[10] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011. doi: 10.5555/1953048.2021068. URL https://dl.acm.org/doi/10.5555/1953048.2021068.

[11] Sai Surya Duvvuri, Fnu Devvrit, Rohan Anil, Cho-Jui Hsieh, and Inderjit S Dhillon. Combining axes preconditioners through kronecker approximation for deep learning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=8j9hz8DVi8.

[12] Runa Eschenhagen, Alexander Immer, Richard E Turner, Frank Schneider, and Philipp Hennig. Kronecker-factored approximate curvature for modern neural network architectures. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Ex3oJEKS53.

[13] Kaixin Gao, Xiaolei Liu, Zhenghai Huang, Min Wang, Zidong Wang, Dachuan Xu, and Fan Yu. A trace-restricted kronecker-factored approximation to natural gradient. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9):7519–7527, May 2021. doi: 10.1609/aaai.v35i9.16921. URL https://ojs.aaai.org/index.php/AAAI/article/view/16921.

[14] Google Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf, 2024. [Online; accessed 19-May-2024].

[15] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 9573–9583, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/48000647b315f6f00f913caa757a70b3-Abstract.html.

[16] Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, et al. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*, 2024.

[17] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[18] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1837–1845. PMLR, 2018. URL http://proceedings.mlr.press/v80/gupta18a.html.

[19] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[20] Satoki Ishikawa and Rio Yokota. When does second-order optimization speed up training? In *The Second Tiny Papers Track at ICLR 2024*, 2024. URL https://openreview.net/forum?id=NLrfEsSZNb.

[21] 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. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/51f3d6252706100325ddc435ba0ade0e-Abstract-Conference.html.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[23] Xi-Lin Li. Preconditioned stochastic gradient descent. *IEEE Transactions on Neural Networks and Learning Systems*, 29(5):1454–1466, 2018. doi: 10.1109/TNNLS.2017.2672978.

[24] Xi-Lin Li. Stochastic hessian fittings with lie groups, 2024. URL https://arxiv.org/abs/2402.11858.

[25] Wu Lin, Felix Dangel, Runa Eschenhagen, Juhan Bae, Richard E. Turner, and Alireza Makhzani. Can we remove the square-root in adaptive gradient methods? A second-order perspective. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 29949–29973. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/lin24e.html.

[26] Hong Liu, Zhiyuan Li, David Leo Wright Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3xHDeA8Noi.

[27] Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. Adalomo: Low-memory optimization with adaptive learning rate. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12486–12502. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.findings-acl.742.

[28] Kai Lv, Yuqing Yang, Tengxiao Liu, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 8187–8198. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.acl-long.445.

[29] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 735–742, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

[30] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2408–2417, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/martens15.html.

[31] James Martens, Jimmy Ba, and Matt Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HyMTkQZAb.

[32] Depen Morwani, Itai Shapira, Nikhil Vyas, Eran Malach, Sham Kakade, and Lucas Janson. A new perspective on shampoo's preconditioner. *arXiv preprint arXiv:2406.17748*, 2024.

[33] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12351–12359, 2019. doi: 10.1109/CVPR.2019.01264.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[35] Abel Peirson, Ehsan Amid, Yatong Chen, Vladimir Feinberg, Manfred K Warmuth, and Rohan Anil. Fishy: Layerwise fisher approximation for higher-order neural network optimization. In *Has it Trained Yet? NeurIPS 2022 Workshop*, 2022. URL https://openreview.net/forum?id=cScb-RrBQC.

[36] Omead Pooladzandi and Xi-Lin Li. Curvature-informed SGD via general purpose lie-group preconditioners, 2024. URL https://openreview.net/forum?id=sawjxRnVpF.

[37] Tomer Porian, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon. Resolving discrepancies in compute-optimal scaling of language models. *CoRR*, abs/2406.19146, 2024. doi: 10.48550/ARXIV.2406.19146. URL https://doi.org/10.48550/arXiv.2406.19146.

[38] Constantin Octavian Puiu. Randomized k-facs: Speeding up k-fac with randomized numerical linear algebra. In Hujun Yin, David Camacho, and Peter Tino, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2022*, pages 411–422, Cham, 2022. Springer International Publishing. ISBN 978-3-031-21753-1.

[39] Constantin Octavian Puiu. Brand new k-facs: Speeding up k-fac with online decomposition updates, 2023. URL https://arxiv.org/abs/2210.08494.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[41] Yi Ren and Donald Goldfarb. Tensor normal training for deep learning models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 26040–26052. Curran Associates, Inc.,

2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/dae3312c4c6c7000a37ecfb7b0aeb0e4-Paper.pdf.

[42] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4603–4611. PMLR, 2018. URL http://proceedings.mlr.press/v80/shazeer18a.html.

[43] Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. A distributed data-parallel pytorch implementation of the distributed shampoo optimizer for training neural networks at-scale. *CoRR*, abs/2309.06497, 2023. doi: 10.48550/ARXIV.2309.06497. URL https://doi.org/10.48550/arXiv.2309.06497.

[44] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[45] Nikhil Vyas, Depen Morwani, and Sham M. Kakade. Adamem: Memory efficient momentum for adafactor. In *2nd Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ICML 2024)*, 2024. URL https://openreview.net/forum?id=fZqMVTz7K5.

[46] Sike Wang, Jia Li, Pan Zhou, and Hua Huang. 4-bit shampoo for memory-efficient network training. *CoRR*, abs/2405.18144, 2024. doi: 10.48550/ARXIV.2405.18144. URL https://doi.org/10.48550/arXiv.2405.18144.

[47] Mitchell Wortsman, Peter J Liu, Lechao Xiao, Katie E Everett, Alexander A Alemi, Ben Adlam, John D Co-Reyes, Izzeddin Gur, Abhishek Kumar, Roman Novak, Jeffrey Pennington, Jascha Sohl-Dickstein, Kelvin Xu, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Small-scale proxies for large-scale transformer training instabilities. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=d8w0pmvXbZ.

[48] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 1204–1213. IEEE, 2022. doi: 10.1109/CVPR52688.2022.01179. URL https://doi.org/10.1109/CVPR52688.2022.01179.

[49] Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George E. Dahl, Christopher J. Shallue, and Roger B. Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8194–8205, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/e0eacd983971634327ae1819ea8b6214-Abstract.html.

[50] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection. *CoRR*, abs/2403.03507,

2024. doi: 10.48550/ARXIV.2403.03507. URL https://doi.org/10.48550/arXiv.2403.03507.

[51] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. (code) galore: Memory-efficient LLM training by gradient low-rank projection. https://github.com/jiaweizzhao/GaLore, 2024.

[52] Rosie Zhao, Depen Morwani, David Brandfonbrener, Nikhil Vyas, and Sham M. Kakade. Deconstructing what makes a good optimizer for language models. *CoRR*, abs/2407.07972, 2024. doi: 10.48550/ARXIV.2407.07972. URL https://doi.org/10.48550/arXiv.2407.07972.

Appendix A. Related Work

We begin by discussing works that are closely related, including Anil et al. [1], George et al. [15] and Zhao et al. [50]. Subsequently, we cover extended related works.

KFAC [30] is a well-known second-order optimization algorithm designed for neural networks. E-KFAC [15] builds upon KFAC in a manner analogous to our extension of Shampoo, introducing a diagonal preconditioner that is updated between KFAC inversion steps. However, E-KFAC's algorithm is not identical to running Adam in KFAC's eigenbasis, as the diagonal preconditioner is not Adam.

Anil et al. [1] introduced several algorithmic and numerical improvements to develop a practical and scalable version of Shampoo [18]. Notably, they empirically found that using an exponent of $1/2$ outperforms the original exponent of $1/4$ in Shampoo. Of particular interest to our work is Appendix B of Anil et al. [1], where, inspired by E-KFAC, they describe an algorithm that is essentially equivalent to SOAP for 2D layers. However, no experiments were provided, and the authors claimed that unpublished experiments showed no empirical improvement over Shampoo. This discrepancy between our findings may be due to some of the implementation details of SOAP.

GaLore [50] was recently proposed as a method to reduce Adam's memory footprint by maintaining momentum in a low-rank subspace derived from the singular value decomposition (SVD) of the gradients. Their algorithm's full-rank version bears similarity to ours, with some notable distinctions. Firstly, their projection subspace is determined by the SVD of the current gradient, while we maintain an exponential moving average of GG^T and G^TG. Secondly, we retain momentum in the original space and project it onto the preconditioned space, whereas they maintain it in the preconditioned space and do not rotate it each time the preconditioned space is updated. In Appendix G, we study GaLore's performance and find that our modifications are necessary for improving upon Shampoo. Moreover, their method only projects one side of a layer using the eigenbasis while using the identity basis on the other side. We examine the impact of one-sided projection for SOAP in Appendix F.1.

Diagonal Preconditioning based Optimizers: Other than AdamW, there are other optimizers which involve diagonal preconditoning such as Lion [3], Sophia [26], and Adafactor [42]. Recent works of Kaddour et al. [21], Zhao et al. [52] showed that these optimizers perform comparably to AdamW for LLM pretraining but do not surpass it. This suggests the need to explore non-diagonal preconditioners. We discuss prior works on non-diagonal preconditioners below.

Second-Order Optimization: Research on second-order optimization in deep learning is generally divided into two categories: Hessian-free methods and methods that estimate the Hessian.

Hessian-Free Methods: Hessian-free approaches [29, 30] optimize without explicitly computing the Hessian matrix, instead employing iterative techniques to approximate the Newton step. Other recent works [23, 24, 36] have focused on designing iterative preconditioners to improve the convergence specifically for stochastic optimization algorithms.

Hessian Estimation Methods: These methods maintain an efficient approximation of the Hessian for neural networks. KFAC [30] and Shampoo [18] are two widely recognized methods in this area.

KFAC [30] was one of the first approaches to go beyond diagonal preconditioners in neural networks, demonstrating that a layer-wise Kronecker-factored preconditioner approximates the layer-wise Hessian in multi-layer perceptrons (MLPs). Subsequent works [31, 33] extended KFAC to other architectures. Recent research [13, 15] has further improved trace and diagonal estimates for KFAC. Efforts to scale up KFAC [2, 12, 38, 39] have focused on making the inversion step more efficient or enhancing distributed implementations.

Shampoo [18], another second-order optimization algorithm, is motivated by the online learning algorithm Adagrad [9]. Shampoo also employs a layer-wise Kronecker-factored preconditioner. A recent

distributed implementation of Shampoo [43] won an optimization efficiency benchmark [4], highlighting the practical utility of second-order methods in deep learning. Few recent works [11, 32] have provided theoretical advancements on top of Shampoo. Other works [1, 25, 35, 46] have proposed various strategies to improve Shampoo's scalability. We defer a comparison of SOAP with these methods to future work.

Appendix B. Theoretical Results

As introduced in Section 3.1, we make three modifications to the Shampoo algorithm, yielding Algorithm 2 (first occurrence of these changes is highlighted in red in the algorithm below). We also formally define an algorithm which can be interpreted as running Adafactor in Shampoo's eigenspace in Algorithm 3.

Algorithm 2 Single step of idealized Shampoo with power $1/2$.

1: Sample batch B_t.
2: $G_t \in \mathbb{R}^{m\times n} \leftarrow -\nabla_W \phi_{B_t}(W_t)$
3: $L \leftarrow \mathbb{E}_B[G_B G_B^T]$ {Where the expectation is over a random batch B.}
4: $R \leftarrow \mathbb{E}_B[G_B^T G_B]$
5: $\hat{H} \leftarrow L \otimes R / \text{Trace}(L)$
6: $W_t \leftarrow W_{t-1} - \eta \hat{H}^{-1/2} G_t = W_{t-1} - \eta L^{-1/2} G_t R^{-1/2} / \text{Trace}(L)^{-1/2}$

Algorithm 3 Single step of idealized Adafactor in Shampoo's eigenspace.

1: Sample batch B_t.
2: $G_t \in \mathbb{R}^{m\times n} \leftarrow -\nabla_W \phi_{B_t}(W_t)$
3: $L \leftarrow \mathbb{E}_B[G_B G_B^T]$
4: $R \leftarrow \mathbb{E}_B[G_B^T G_B]$
5: $Q_L \leftarrow \texttt{Eigenvectors}(L)$
6: $Q_R \leftarrow \texttt{Eigenvectors}(R)$
7: $G'_t \leftarrow Q_L^T G_t Q_R$
8: {Idealized version of code for Adafactor taking G'_t to be the gradient}
9: $G'_{B_t} \leftarrow Q_L^T G_{B_t} Q_R$
10: $A = \mathbb{E}_B[G'_B \odot G'_B]\mathbf{1}_m$ where $G'_B = Q_L^T G_B Q_R$
11: $C = \mathbf{1}_n^\top \mathbb{E}_B[G'_B \odot G'_B]$
12: $\hat{V}_t = \frac{AC^T}{\mathbf{1}_n^\top A}$ {Elementwise division}
13: $G''_t \leftarrow \frac{G'_t}{\sqrt{\hat{V}_t}+\epsilon}$ {Elementwise division and square root}
14: $G'''_t \leftarrow Q_L^T G''_t Q_R$ {Projecting back to original space}
15: $W_t \leftarrow W_{t-1} - \eta G'''_t$

Claim 2 *Algorithms 2 and 3 are equivalent.*

Proof

Consider G_t in the basis created after rotating by Q_L, Q_R i.e. $G'_t = Q_L^T G_t Q_R$. Let the eigenvalues of $\mathbb{E}_{B_t}[G_{B_t} G_{B_t}^T]$ and $\mathbb{E}_{B_t}[G_{B_t}^T G_{B_t}]$ be given by $\lambda_1, ..., \lambda_m$ and $\mu_1, ..., \mu_n$ respectively. Algorithm 1 scales the i, j coordinate by $(\lambda_i \mu_j / (\sum_i \lambda_i))^{-1/2}$, while Algorithm 2 scales them by $(A_i C_j / (\sum_i A_i))^{-1/2}$. We now show that $A_i = \lambda_i$, an analogous argument shows $C_j = \mu_j$.

$$
\begin{aligned}
A_i &= e_i^T \mathbb{E}_B[G'_B \odot G'_B]\mathbf{1}_m \\
&= \mathbb{E}_B[\sum_j (G'_B)^2_{i,j}] \\
&= \mathbb{E}_B[\sum_j (u_i^T (G_B) v_j)^2] && \text{(Using definition of } G'\text{)} \\
&= \mathbb{E}_B[||u_i^T (G_B)||^2] && (v_j \text{ form a basis}) \\
&= \mathbb{E}_B[u_i^T G_B G_B^T u_i] \\
&= \lambda_i && \text{(By definition of } \lambda_i \text{ and } u_i\text{)}
\end{aligned}
$$

■

Appendix C. Further Implementation Details of SOAP

In Section 3.1, we introduced SOAP and Algorithm 1 which runs Adam in Shampoo's eigenspace. Here we describe some additional implementation details:

1. Algorithm 1 describes the behavior of the algorithm for 2D layers. Following Zhao et al. [50], for 1D layers we run standard AdamW. This reduces the overhead as compared to standard implementations of Shampoo which solve an eigenvector problem for 1D layers too.

2. Following Wang et al. [46], we compute eigenvectors of L (and R) using one step of power method (Algorithm 4). This requires doing one matrix multiplication followed by QR decomposition. QR decomposition is faster [8] than standard eigenvector decomposition in PyTorch. For the first iteration, eigenvectors are initialized by doing a standard eigenvector decomposition.

3. For layers with huge dimensions such as the first and last layer in language modeling transformers, maintaining the eigenvectors would be space and time prohibitive. For such dimensions we fix the rotation matrix (Q_L or Q_R) to be identity. Note that if we fix both Q_L and Q_R to be identity for a 2D layer, we would recover Adam.

4. Algorithm 1 omits bias correction and weight decay for simplicity, but these are used in the actual implementation, identical to their use in AdamW.

Algorithm 4 `Eigenvectors` function, implemented using power iteration and QR decomposition. Inputs: PSD matrix P and estimate of eigenvectors Q. If the estimate was exact we would have $P = QDQ^T$ where D is the diagonal matrix with eigenvalues.

1: $S \leftarrow PQ$
2: $Q \leftarrow \texttt{QR}(S)$

Appendix D. Experimental Setup

Many aspects of our setup such as models are the same as in Zhao et al. [52]. We will restate those details verbatim for completeness.

We train language models on C4 tokenized with the T5 tokenizer [40] and report results in terms of validation loss.

Models. We start from the OLMo codebase [16] and train decoder-only transformer models of three sizes: 210m, 360m, and 660m, where the parameter count refers to non-embedding parameters. The models have widths of 1024, 1024, and 1408 and depths of 12, 24, 24. We used the 210m model to explore various ablations, most of our reported results are on 360m and 660m. The MLP hidden dimension is 4x of the width. The activation function is GeLU [19]. We use RoPE positional encodings [44]. Attention heads are always dimension 64. We use PyTorch default LayerNorm. We use QK layer norm [6]. Following Wortsman et al. [47] we do not learn biases for the linear layers or LayerNorms. We train in mixed precision with bfloat16.

Algorithms. We use the standard Pytorch implementation of AdamW [34], the DistributedShampoo [43] implementation of Shampoo. We implement ourselves SOAP and GaLore starting from an older version of Pytorch implementation of AdamW and the official GaLore implementation [51].

Default hyperparameters. We use $\beta_1 = 0.95$, as we found it to outperform $\beta_1 = 0.9$ in our sweeps for the 360m model. Following Wortsman et al. [47] we use decoupled weight decay with coefficient $1\text{e}{-4}$ and z-loss with coefficient $1\text{e}{-4}$. We use the default value of $\epsilon = 1e-8$ in AdamW (actual or when used for grafting), SOAP and GaLore. We use warmup followed by cosine decay as our scheduler. We start the warmup and end the cosine decay at $0.1x$ the maximum learning rate.

Default hyperparameters for DistributedShampoo Shi et al. [43] state that they find the optimal exponent to be either $-1/2$ or $-1.82/4 \approx -1/2.2$. Our preliminary findings were similar to this. Hence we set the default values of exponent to be $-1/2.5$ for both 1D and 2D parameters. We set $\epsilon_{\text{shampoo}} = 1\text{e}{-12}$ and $\beta_{\text{shampoo}} = 0.95$ based on our initial set of experiments on the 210m model.

Default hyperparameters for GaLore GaLore introduces an additional hyperparameter called scale (α) since due to low rank updates the overall update magnitude decreases. Since we are running a full rank version of GaLore we set $\alpha = 1$.

Token counts. For all of our runs we use a sequence length of 1024. For all models (except in Appendix E.2), we use a token batch size of 2048k $\approx$ 2m. We default to training models for the approximately "chinchilla optimal" number of tokens that is $\approx$20 times the number of parameters. Explicitly, this means for our default batch size of 2m, the 210m models are trained for 1600 steps or $\approx$ 3.3b tokens. The 360m models are trained for 3200 steps, the 660m models are trained for 6400 steps.

D.1. Sweeping over hyperparameters

AdamW, 2m batch size: Starting from the default hyperparameters above we do the following sweeps:

1. We sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.
2. (360m) We sweep over the cross product of best 3 learning rates and $\beta_1 \in \{0.9, 0.95, 0.99\}$.
3. (360m) We sweep over the cross product of best 3 learning rates and $\beta_2 \in \{0.9, 0.95, 0.99\}$.

The last two of the sweeps did not yield any benefit for the 360m model with 2m batch size hence we only sweep over learning rate for the 660m model with 2m batch size.

DistributedShampoo, 2m batch size: Starting from the default hyperparameters above we do the following sweeps:

1. We sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.
2. (360m) We sweep over over the cross product of best 3 learning rates from above and $\epsilon_{\text{shampoo}} \in \{1\text{e}{-11}, 1\text{e}{-12}, 1\text{e}{-13}\}$.
3. (360m) We sweep over over the cross product of best 3 learning rates from above and $\beta_{\text{shampoo}} \in \{.9, .95, .975\}$.
4. Let e_1, e_2 denote the exponents used in DistributedShampoo for 1D and 2D parameters respectively. We also sweep over the cross product of best 3 learning rates from above and (e_1, e_2) in $\{(2, 2), (2.5, 2.5), (3, 3), (2, 4)\}$.

These sweeps did not yield any significant improvement in performance ($< .004$) for the 360m model. Hence we only sweep over the learning rate for the 660m model.

SOAP, 2m batch size: Starting from the default hyperparameters above we sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.

AdamW, 256k batch size: For the 360m model with 256 batch size we start from the default hyperparameters and do the following sweeps:

1. We sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.
2. We sweep over the cross product of best 3 learning rates and $\beta_2 \in \{0.95, 0.99\}$.

In the second sweep we observe small improvements in performance by using $\beta_2 = .99$, so our final numbers use $\beta_2 = .99$. This (small) improvement in performance by using a larger β_2 at smaller batch sizes was also observed by Porian et al. [37], Zhao et al. [52].

DistributedShampoo, 256k batch size: For the 360m model with 256 batch size we start from the default hyperparameters and do the following sweeps:

1. We sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.
2. We sweep over the cross product of best 3 learning rates and $(\beta_2, \beta_{\text{shampoo}}) \in \{(.95, .95), (.99, .99)\}$.

In the second sweep we observe small improvements in performance by using $\beta_2 = \beta_{\text{shampoo}} = .99$, so our final numbers use $\beta_2 = \beta_{\text{shampoo}} = .99$.

SOAP, 256k batch size: For the 360m model with 256 batch size we start from the default hyperparameters and do the following sweeps:

1. We sweep over learning rate in $\{.1, .0316, .01, \ldots, 3.16\text{e}{-4}\}$.
2. We sweep over the cross product of best 3 learning rates and $\beta_2 \in \{.95, .99\}$.

In the second sweep we observe small improvements in performance by using $\beta_2 = .99$, so our final numbers use $\beta_2 = .99$.

Preconditioning frequency sweeps: For the preconditioning frequency experiments of SOAP and Shampoo (Figure 1 (right)), for each frequency we do a learning rate sweep over the best 3 learning rates

found at preconditioning frequency 10. Other hyperparameters are set to their optimal values obtained using the precondition frequency 10 sweeps.

360m and 660m shorter runs: For each of the shorter runs of 360m and 660m models for the SOAP optimizer (Figure 2), we did learning rate sweep over the best 3 learning rates found for the standard length run. Other hyperparameters are set to their optimal values obtained using the standard length run.

Warmup: The warmup duration for the 360m and 660m models were 600 and 1200 steps respectively. For the shorter runs (Figure 2), for 360m model, the warmup durations were 400, 400, 500 and 525 steps for 0.5, 0.625, 0.75 and 0.875 runs respectively. For the 660m model, the warmup durations were 600, 750, 900 and 1050 steps for 0.5, 0.625, 0.75 and 0.875 runs respectively. For 360m model with 256k batch size (Appendix E.2) we use a warmup for 4000 steps (total steps is 25000).

Appendix E. Additional Experimental Results

E.1. Efficiency Benefits

In this section, we provide additional results to accompany our main findings from Sections 5.1 and 5.2. In Figure 2, we provide the precise efficiency benefit calculations after fitting a scaling law across multiple SOAP runs on varying fractions of the data. In Figure 3 we provide analogous results from our main Figure 1 for 360m models.

E.2. Effect of Batch Size

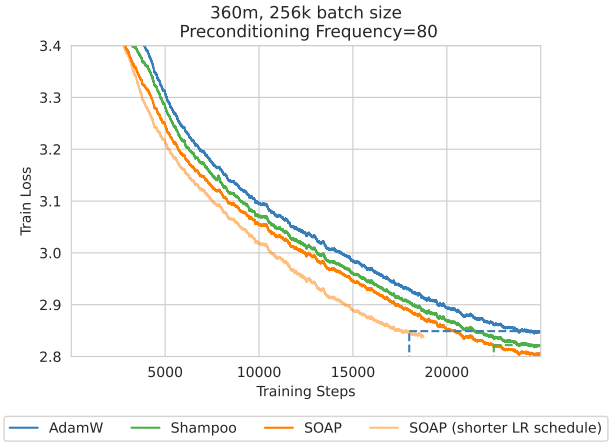

Figure 4: Comparing performance of tuned runs for AdamW, Shampoo (using DistributedShampoo [43] implementation) and SOAP for token batch size of 256k. Shampoo and SOAP use preconditioning frequency of 80. We observe a $\geq 25\%$ reduction in the number of iterations compared to AdamW, and approximately a 10% reduction compared to Shampoo. See Figure 2 (right) for wall-clock time improvement and Section 4 for detailed calculation of efficiency improvement.

In this section, we examine the impact of batch size on the performance of the Shampoo and SOAP optimizers. Specifically, we reduce the batch size by a factor of 8, from 2m to 256k. To maintain the same FLOPS overhead for the eigenvector decomposition steps as in the 2m setting, we increase the preconditioning frequency by a factor of 8, from 10 to 80. In Figure 4, we present the optimal runs for each optimizer. Our results show that SOAP consistently outperforms both Shampoo and AdamW, demonstrating a reduction of

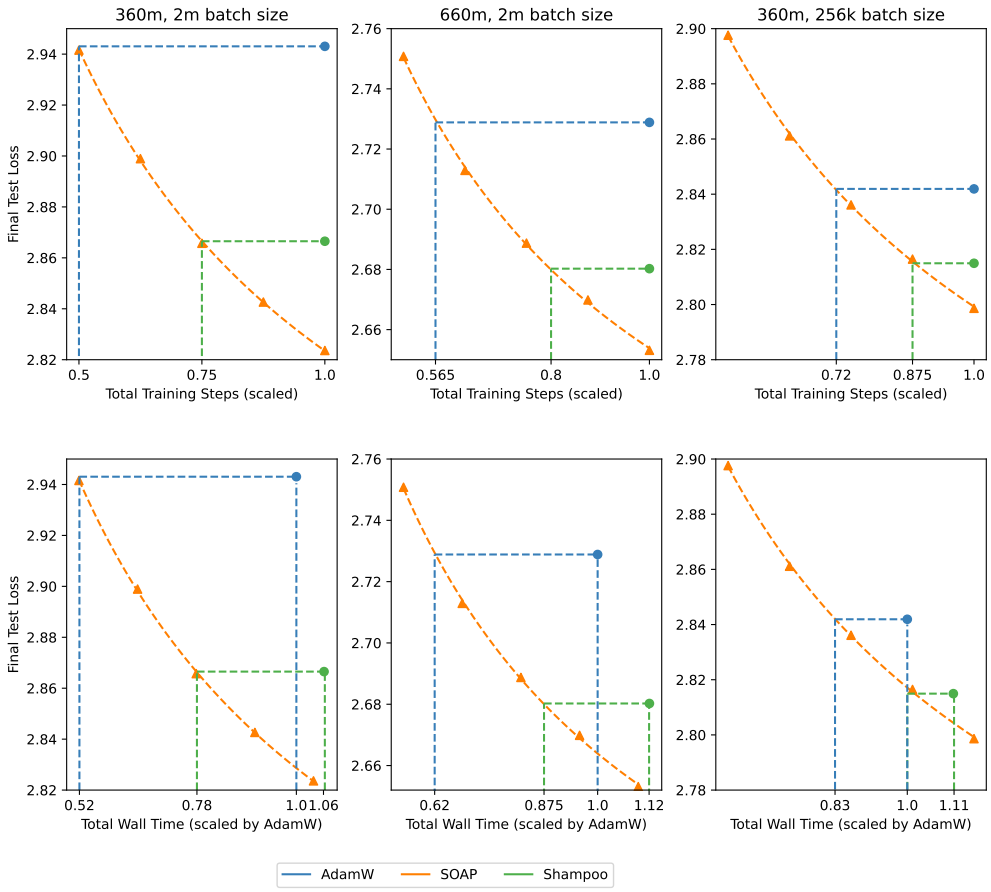

Figure 2: Precise efficiency benefits of SOAP over AdamW and Shampoo for 360m (at 256k and 2m batch size) and 660m (at 2m batch size) model. For the precise methodology, refer to Section 4.

25% or more in the number of iterations compared to AdamW, and approximately a 10% reduction compared to Shampoo. In Figure 2 (right), we show that SOAP also improves in wall-clock time by $\geq 15\%$ over AdamW and approximately 10% over Shampoo. Note that we present these results as a preliminary analysis for small batch size runs. It is quite likely that our increase in preconditioning frequency by a factor of 8 is not optimal and a better trade-off is achievable. Furthermore, the overhead of SOAP can likely be ameliorated by doing L and R updates in lower precision (instead of fp32).

We also note that the decrease in efficiency improvements at smaller batch sizes for second-order methods is consistent with prior works [20, 49].

Appendix F. Further Efficiency Improvements for SOAP

In this section, we discuss space and time complexity of SOAP and provide an overview of potential avenues for further space and compute efficiency improvements in SOAP.

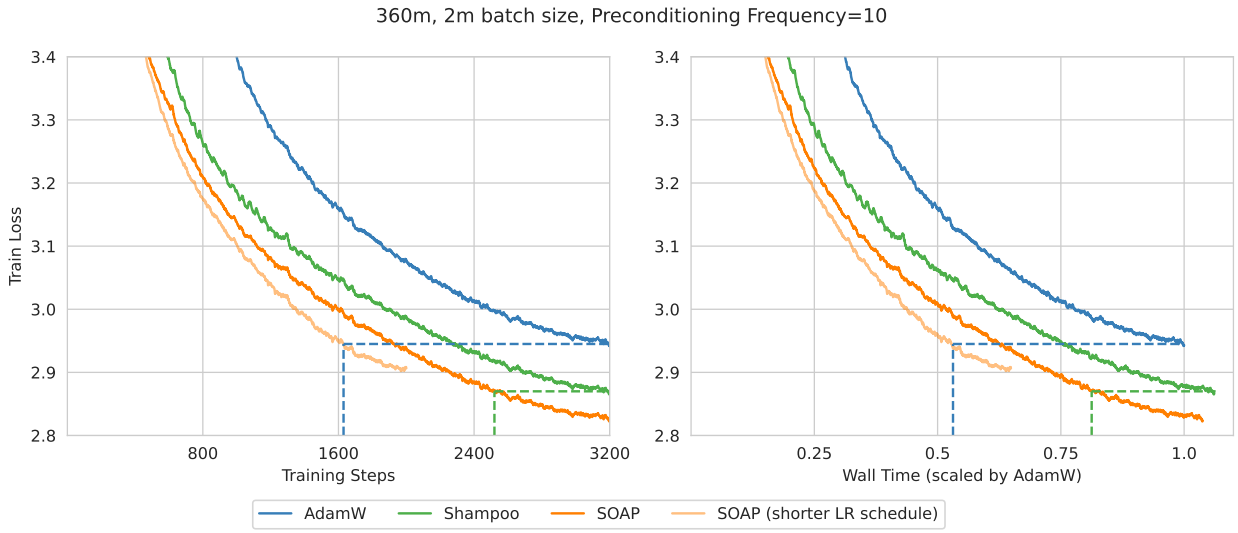

Figure 3: Comparing performance of tuned runs for AdamW, Shampoo (using DistributedShampoo [43] implementation) and SOAP. Shampoo and SOAP use preconditioning frequency of 10. We observe a $\geq 40\%$ reduction in the number of iterations and a $\geq 35\%$ reduction in wall clock time compared to AdamW, and approximately a 20% reduction in both metrics compared to Shampoo. See Figure 1 for 660m results, Section 5.2 and appendix E.2 for ablations of preconditioning frequency and batch size respectively, and Section 4 for detailed calculation of efficiency improvement and experimental methodology.

F.1. One Sided Eigenbasis

As described in Appendix A, Zhao et al. [50] have an algorithm similar to ours. One of the differences is that they only project the smaller side of the layer using the eigenbasis while using identity as the rotation matrix for the larger side i.e. if $m < n$ we set $Q_R = I_n$ in Algorithm 1 and if $m > n$ we set $Q_L = I_m$. Doing this leads to a reduction in space usage as well as reduction of optimizer time overhead, which is discussed in Appendix F.2.1 and Figure 3.

In Figure 5, it is evident that the one-sided projection results in slightly reduced performance compared to the original SOAP optimizer. However, it still performs on par with, or marginally better than, Shampoo, while maintaining greater computational efficiency. Further investigation into the potential for these variants to surpass the computational efficiency of original SOAP optimizer is left for future work.

F.2. Space usage of SOAP

For a $m \times n$ matrix where $m > n$ we require

$$2m^2 \text{ (for } L, Q_L) + 2n^2 \text{ (for } R, Q_R) + 3mn \text{ (for gradient, } M, V)$$

space usage[4] (beyond weights and activations), specifically for L, Q_L, R, Q_R, momentum (M), AdamW's second order estimate (V), and the gradient. This is the same space usage as DistributedShampoo while AdamW uses $3mn$.

4. One mn is for storing the gradients, this can be avoided (as long as there is no gradient accumulation) by applying gradients along with backprop [28] but this is not implemented by default in standard deep learning frameworks such as PyTorch. Hence we will include this term in all of our calculations.

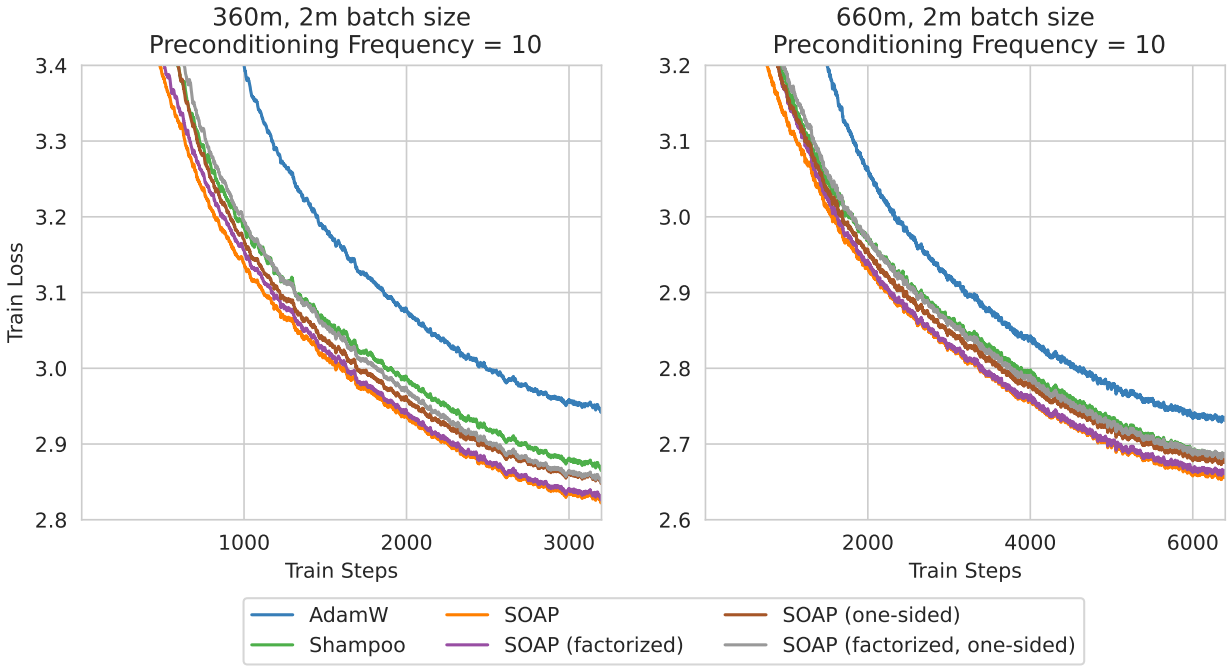

Figure 5: Performance of variants of SOAP which improve space usage or time overhead. 1. SOAP (factorized): Uses Adafactor instead of Adam in Shampoo's eigenbasis and 2. SOAP (one-sided): Uses $Q = I$ (i.e. no rotation) on the large side of weight matrix and 3. SOAP (factorized, one-sided): Combines both of these changes. We observe that while using Adafactor instead of Adam causes a negligible increase in loss, using the one-sided variant causes a larger increase. However, the one-sided variant also has much larger reduction in time and space overhead. For computational benefits of these variants see Appendices F.2 and F.3.

F.2.1. Improving space usage of SOAP

The most direct way to reduce memory is using low precision to store the L, R, Q_L, Q_R, V matrices, which is done by Dettmers et al. [7], Wang et al. [46]. Orthogonal to the low precision approaches, there are two algorithmic approaches to improving the space usage of SOAP:

- Using Adafactor instead of Adam as the diagonal preconditioner after rotating by Q_L and Q_R. This reduces the space usage by mn.
- Using one sided version of SOAP (Appendix F.1). This reduces space usage from $2m^2 + 2n^2 + 3mn$ to $2\min(m, n)^2 + 3mn$.
- Combining these approaches yields space usage of $2\min(m, n)^2 + 2mn$.

For standard transformer architectures the last variant which combines the two approaches would yield less space usage overall compared to AdamW (which uses $3mn$).

We try these approaches in Figure 5. We observe that using Adafactor instead of AdamW yields very small reductions in performance while using one-sided preconditioner results in larger reductions. Nonetheless even after combining these two approaches the resulting optimizer outperforms AdamW while having a smaller space requirement than AdamW. Regarding space usage we also note that Adafactor (with momentum added back) itself utilizes only $2mn$ space usage and has been shown to perform comparable to AdamW

for ViT training [48] and for language model training [52]. Further space reduction beyond Adafactor has been studied in the Adalomo [27], GaLore [50], and AdaMeM [45] papers.

F.3. Time Overhead of SOAP

There are two types of overhead of Shampoo and SOAP over AdamW: the overhead per step and the overhead when changing the preconditioner (or for SOAP, the preconditioner's eigenbasis). Let us first analyze the first one. For SOAP per step for a layer of size $m \times n$ we have an overhead of

$$m^3 \text{ (updating } L) + n^3 \text{ (updating } R) + (2m^2n + 2mn^2) \text{ (projecting and projecting back on both sides).}$$

We note that this is more than the overhead of Shampoo which is $m^3 + n^3 + m^2n + n^2m$. This can be observed in Figure 2 (bottom, right) but not in the other figures since there the second type of overhead is the dominant term.

The second type of overhead is due to changing the preconditioner for Shampoo (or for SOAP, preconditioner's eigenbasis i.e. Q_L and Q_R). The DistributedShampoo [43] implementation of Shampoo uses a direct call to `torch.linalg.eigh` for this. Following Wang et al. [46] we use Algorithm 4 which uses power iteration based approach which calls `torch.linalg.qr`. We note that `torch.linalg.qr` is faster than `torch.linalg.eigh` [8]. In Figure 6 (right) we see that using power iteration based approach (`torch.linalg.qr`) performs as well as fresh eigenvector decomposition (`torch.linalg.eigh`).

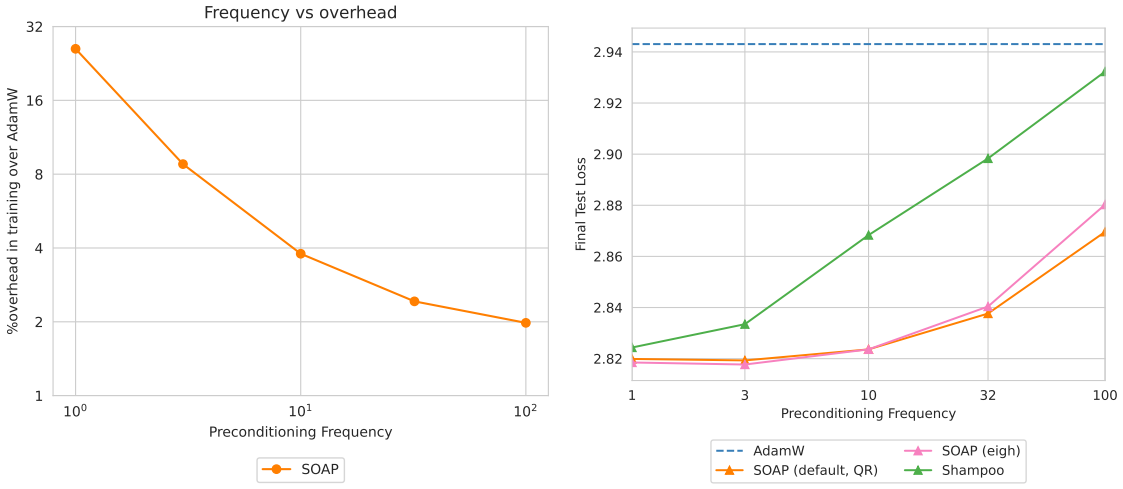

Figure 6: (Left) Depicting the overhead of SOAP over AdamW as a function of preconditioning frequency (Right) Comparing the performance of SOAP with `torch.linalg.eigh` for computing the eigenvectors with Algorithm 4, which uses `torch.linalg.qr`. Note that `torch.linalg.qr` is computationally more efficient than `torch.linalg.eigh` (as mentioned in Documentation [8]); however, both seem to have comparable performance throughout the preconditioning frequency spectrum.

Effect of frequency on overhead: In Figure 6 (left), we observe that the overhead decreases as the preconditioning frequency increases, i.e., the frequency of invoking Algorithm 4. If the only additional computation occurred in Algorithm 4, we would expect the overhead to scale as $1.0/(\text{preconditioning frequency})$, approaching zero. However, empirical results (Figure 6 left) show that the overhead approaches an asymptote greater than zero. This is attributable to the additional matrix multiplications required to update L,

update R, project the gradient, and reproject the gradient (for each layer) in the optimizer. Currently, these operations are performed in float32; reducing the precision of these operations, as proposed in [46], could lower this asymptote.

F.3.1. IMPROVING TIME OVERHEAD OF SOAP

The per step overhead of SOAP can be reduced by using low precision to store the L, R, Q_L, Q_R, V matrices [7, 46], which in turn will speed up computation done using these matrices. This approach cannot be used for reducing the overhead for the preconditioner update in popular deep learning frameworks such as Pytorch since `torch.linalg.qr` does not support precision lower than `float32`. Orthogonal to the low precision approach we can improve the per step time overhead of SOAP by the following algorithmic approaches:

- Using Adafactor instead of Adam (Appendix F.2) as the diagonal preconditioner after rotating by Q_L and Q_R. In this version of SOAP the overhead can be improved by from $m^3 + n^3 + 2m^2n + 2n^2m$ to $m^3 + n^3 + m^2n + n^2m + \max(m, n)^2 \min(m, n) + \min(m, n)^3$ by merging the project and project back steps for the smaller dimension.
- Using one sided version of SOAP (Appendix F.1). This reduces overhead from $m^3 + n^3 + 2m^2n + 2n^2m$ to $\min(m, n)^3 + 2\min(m, n)^2 \max(m, n)$.
- Combining these approaches yields an overhead of $\min(m, n)^2 \max(m, n) + 2\min(m, n)^3$

Using one-sided version also reduces the second type of overhead from a calls to `torch.linalg.qr` on a $m \times m$ and a $n \times n$ matrix to only a single call to $\min(m, n) \times \min(m, n)$ matrix.

Appendix G. GaLore

We tried GaLore for 210m model, and while it outperformed AdamW it performed worse than Shampoo. Hence we do not try GaLore for higher model sizes.

Hyperparameter sweeps: We did the following sweeps:

1. We swept the cross product over learning rate $(3.16e-4, 1e-3, 3.16e-3, 1e-2)$, preconditioning frequency $(10, 50, 200)$, both sided and one sided versions. Frequency 200 had the best results matching the observation of Zhao et al. [50].
2. We did a cross product sweep over learning rate $(3.16e-4, 1e-3, 3.16e-3, 1e-2)$, both sided and one sided versions with $\beta_2 = .99$ instead of .95 and preconditioning frequency 200.
3. We did a cross product sweep over learning rate $(3.16e-4, 1e-3, 3.16e-3, 1e-2)$, both sided and one sided versions, preconditioning frequency $(50, 200)$ with $\beta_1 = .9$ instead of .95.

The best performing run among all of these achieved a final loss of 3.12 while the best Shampoo run achieved a final loss of 3.10.