AdaMeM: Memory Efficient Momentum for Adafactor

Nikhil Vyas [*1] **Depen Morwani** [*1 2] **Sham Kakade** [1 2]

Abstract

Adafactor is a memory efficient algorithm which does not maintain momentum and has near 0 memory overhead as compared to gradient descent. However it performs worse than Adam in many setups. Prior works have shown that this gap can be removed by adding momentum to Adafactor. This comes at the cost of increased memory requirements. In this work we use the ideas of low rank optimizers such as LoRA and GaLore to maintain momentum on a low rank subspace of the weights on top of Adafactor to give a new optimizer: AdaMeM. However unlike low rank optimizers we still utilize full rank gradients but maintain momentum only on the top SVD subspace of the gradients. We show results on language modelling for models of size 210M and 550M demonstrating improved performance over Adafactor and GaLore. We also give theoretical arguments supporting the design of AdaMeM.

1. Introduction

Adam (Kingma & Ba, 2015), the default optimizer utilized in language modeling tasks within deep learning, is considerably more memory-intensive compared to stochastic gradient descent (SGD) (Robbins & Monro, 1951), as it maintains two additional values: the first and second-order momentum for each network parameter. The development of memory-efficient optimizers that retain Adam's performance remains a critical area of research, as evidenced by numerous previous studies.

Adafactor (Shazeer & Stern, 2018), in its original proposal, entirely eliminated the memory overhead by removing the first-order momentum and implementing a highly efficient factorization of the second-order momentum. Nonetheless, subsequent works have demonstrated that Adafactor's performance lags behind that of Adam (Rae et al., 2021; Zhai et al., 2022). Reintroducing the first-order momentum has been shown to bridge this performance gap (Zhai et al., 2022). Similarly, a recently proposed optimization algorithm, Lion (Chen et al., 2023), also removes the memory overhead associated with the second-order momentum, substituting it with a sign operation.

Other research efforts have focused on reducing the number of parameters involved in the overall update, thereby diminishing the memory overhead for both first and second-order momentum. LoRA (Hu et al., 2022) introduced low-rank updates for fine-tuning network parameters, significantly reducing the optimizer's memory requirements. ReLoRA (Lialin et al., 2023) extended this approach to pretraining by periodically merging the low-rank updates into the parameters. GaLore (Zhao et al., 2024) also maintained low-rank updates for parameters, but specifically within the top singular value decomposition (SVD) subspace of the gradients.

In this work, we integrate the memory-efficient preconditioning of Adafactor with the low-rank momentum concepts of GaLore, resulting in AdaMem—a capable optimizer with substantially lower overhead compared to Adam. We empirically compare our method with Adam, Adafactor, and GaLore for language pretraining tasks, demonstrating that our method, which incorporates low-rank momentum, achieves performance significantly closer to Adam than either Adafactor or GaLore.

We support our empirical findings with theoretical results. Specifically, we establish a novel connection between low-rank optimization and the Shampoo optimizer (Gupta et al., 2018), illustrating that the low-rank subspace used for maintaining momentum approximates the top eigenspace of the Hessian, based on the assumptions of the Shampoo optimizer. For linear regression, we further demonstrate that maintaining momentum in the top eigenspace of the Hessian substantially enhances the convergence speed of gradient descent.

[*]Equal contribution [1]SEAS, Harvard University [2]Kempner Institute, Harvard University. Correspondence to: Nikhil Vyas <nikhil@g.harvard.edu>.

Accepted to the Workshop on Advancing Neural Network Training at International Conference on Machine Learning (WANT@ICML 2024).

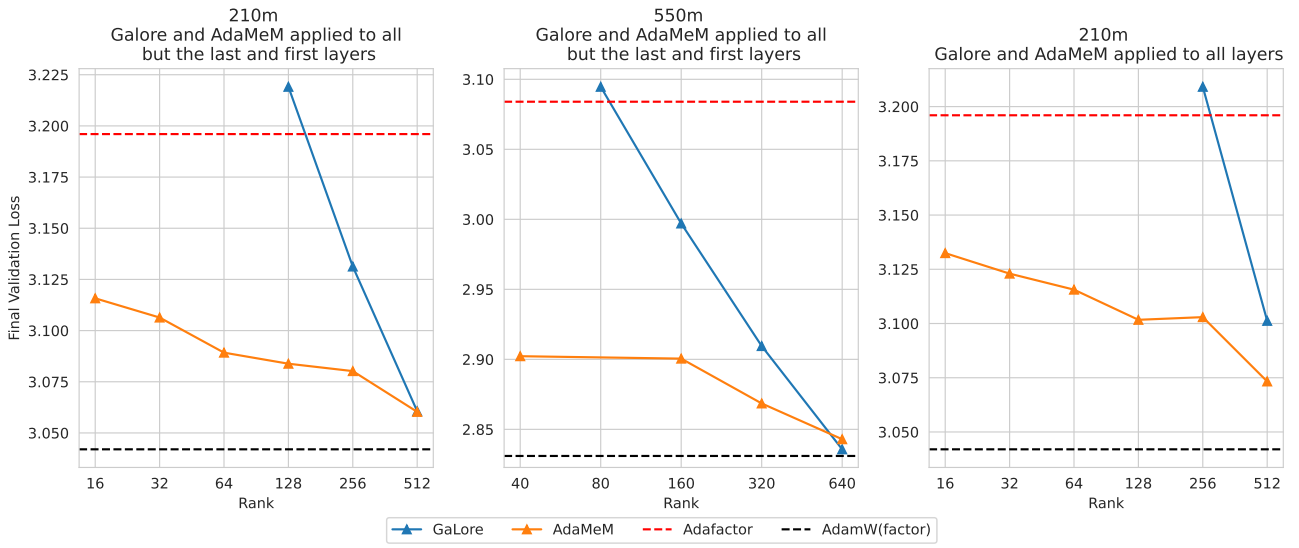

Figure 1. Comparison of our algorithm (AdaMeM) to Adafactor and Adam as baselines and the prior work of GaLore. In these plots we plot final validation loss as a function of rank used by AdaMeM and GaLore. We observe that while AdaMeM is always better than Adafactor, GaLore is only better than Adafactor at large ranks. In the right and middle figure the low rank approximation is applied to all layers except first and last layers while in the last plot it is applied to all layers. See Section 5 for additional details.

2. Related Works

Low-Rank Optimizers: The training of language models using low-rank methods has become a prominent area of research, experiencing a significant surge in recent times. LoRA (Hu et al., 2022) introduced low-rank fine-tuning of large language models (LLMs), demonstrating competitive performance across various downstream tasks. ReLoRA (Lialin et al., 2023) extended this approach to pre-training by periodically merging low-rank updates into the weights. Similarly, GaLore (Zhao et al., 2024) proposed using low-rank updates, specifically along the top SVD subspace of the gradients.

Low-Memory Optimizers: Adafactor (Shazeer & Stern, 2018) is a variant of Adam that maintains factored second moments and does not use momentum. As a result, this optimizer incurs almost no[1] overhead. Given that Adafactor's space requirements are negligible compared to the weights, we will use it as our primary baseline for low-space optimizers. Additionally, the recently proposed optimizer Lion (Chen et al., 2023) reduces the overall memory footprint compared to Adam by replacing the second-order momentum in the Adam update with a sign operation. There is also a large literature (**????**) on optimizers using lower precision to save memory. These approaches based on using lower precision to save memory are orthogonal to low rank based approaches and can be combined with them (Zhao et al., 2024), this also applies to our approach.

Adamw(factor) as a baseline: In some of our experiments, Adafactor with momentum outperforms Adam. Therefore, we will report the best results from these two methods as Adamw(factor), which will serve as our baseline for algorithms utilizing full-rank momentum.

Benefit of using Momentum: Given that Adafactor does not use momentum, it is expected to perform poorly in scenarios where momentum benefits optimization speed. Previous studies (Kidambi et al., 2018; Lee et al., 2022; Wang et al., 2024) have argued and empirically demonstrated that momentum is advantageous with large batch sizes, with its benefits diminishing at smaller batch sizes. We empirically explore this in Section 6, confirming that these observations hold for pretraining language models. This leads to the following question: **With large batch sizes, how can we gain the benefits of momentum without incurring its space cost?**

3. Algorithm

We begin by describing the intuition behind AdaMeM (Algorithm 1). Low-rank algorithms such as LoRA (Hu et al.,

[1]The factored second momentum statistics occupy $m+n$ space for an $m \times n$ layer, which is negligible compared to the mn space required to store the weights themselves. Common implementations in libraries such as PyTorch typically use twice the space of the weights, as they also store gradients for all layers. This can be mitigated by optimizers like LOMO (see Appendix A for further discussion). Throughout this paper we will assume such an implmentation.

2022), ReLoRA (Lialin et al., 2023), and GaLore (Zhao et al., 2024) perform gradient descent with momentum in a low-dimensional space. Our approach builds upon the methodology of GaLore. For a gradient matrix $G \in \mathbb{R}^{m\times n}$ with $m \leq n$, GaLore maintains momentum within the top r rank subspace corresponding to the left singular vectors of G. Specifically, let $P \in \mathbb{R}^{m\times r}$ denote the projection matrix to the top r left singular vectors of G. Momentum is then maintained for P^TG, with P itself being updated every $T = 200$ steps.

Our improvement is based on a straightforward concept: in addition to running gradient descent with momentum in the low-rank space, we can also perform gradient descent (without momentum) on the residual space using the gradient of the current batch. This requires using full-rank gradients while maintaining momentum solely in the low-rank space. Although this outlines the core idea behind our algorithm, further complexities arise because we aim to implement Adam-like algorithms instead of SGD.

We address these complexities as follows: First, for the low-rank component of the gradient, we run Adafactor with momentum in the low-rank space, similar to the Adafactor variant of GaLore. A natural choice for handling the residual gradient would be to apply Adafactor (without momentum) in the complementary space. However, we lack a basis for the complementary space. Consequently, we could opt to run Adafactor (without momentum) for the residual gradient in the original space. This approach has a drawback: while the low-rank gradient and the residual gradient are orthogonal by definition, preconditioning the low-rank gradient in the low-rank space and the residual gradient in the original space with Adafactor can disrupt this orthogonality. To maintain orthogonality, we employ a one-sided version of Adafactor, as described in Algorithm 4. We prove below that this preserves orthogonality. In Section 6, we demonstrate that using the standard Adafactor instead results in suboptimal performance.

Claim 3.1. *Let $G \in \mathbb{R}^{m\times n}$ be the gradient matrix with $m \leq n$ and $P \in \mathbb{R}^{m\times r}$ be the projected matrix. Then preconditionng the projected gradient (P^TG) for Adafactor (Algorithm 3) and residual gradient ($(I - PP^T)G$) by one sided Adafactor (Algorithm 4) preserves their orthogonality.*

Proof. We begin by noting that precondition P^TG by Adafactor (Algorithm 3) results in a matrix of the form $A = PD_1P^TGD_2$ where $D_1 \in \mathbb{R}^{r\times r}, D_2 \in \mathbb{R}^{n\times n}$ are diagonal matrices. We note that $A \in \mathbb{R}^{m\times n}$ is in the original space. Preconditioning residual gradient ($(I - PP^T)G$) by one sided Adafactor (Algorithm 4) results in a matrix of the form $B = (I - PP^T)GD_3$ where $D_3 \in \mathbb{R}^{n\times n}$ is a diagonal matrices.

We are interested in the inner product of A and B when they are treated as vectors i.e. $\mathrm{Tr}(A^TB)$.

$$\mathrm{Tr}(A^TB) = \mathrm{Tr}(D_2^TG^TPD_1^TP^T(I - PP^T)GD_3) = 0$$

since $P^T(I - PP^T) = 0$. □

The pseduocode for AdaMeM is given in Algorithm 1.

Memory Requirements. We note here that our memory requirement (beyond the weights themselves) for maintaining rank r momentum is mr(for projection matrix P) + nr(for momentum in the low rank space) + r + n + n(for Adafactor's second moments) $\approx (m + n)r$ same as the Adafactor version of GaLore.

Algorithm 1 AdaMeM.
Bias correction and regularization constant ϵ not described for simplicity.

Require: A layer weight matrix $W \in \mathbb{R}^{m\times n}$ with $m \leq n$. Step size η, relative step size $\delta = 1$, decay rates $\beta_1 = .9$, $\beta_2 = .95$, rank r, subspace change frequency $T = 200$.
Initialize first-order moment $M_0 \in \mathbb{R}^{n\times r} \leftarrow 0$
Initialize auxiliary variables Z_0, Z_0^{os} to be used by Adafactor preconditioner (Algorithm 3) and one-sided Adafactor preconditioner (Algorithm 4).
Initialize step $t \leftarrow 1$, auxillary
repeat
  $G_t \in \mathbb{R}^{m\times n} \leftarrow -\nabla_W \phi_t(W_t)$
  $R_t \leftarrow$ Project(G_t, r, T, t) {Project gradient into compact space using Algorithm 2}
  $S_t \leftarrow G_t - P_tR_t$ {Residual gradient outside of the compact space}
  $M_t \leftarrow \beta_1 \cdot M_{t-1} + (1 - \beta_1) \cdot R_t$
  $N_{t,1}, Z_t =$ Adafactor-Preconditioner(M_t, Z_{t-1}, β_2)
  $N_{t,2}, Z_t^{\text{os}} =$ OS-Adafactor-Preconditioner$(S_t, Z_{t-1}^{\text{os}}, \beta_2)$
  $W_t \leftarrow W_{t-1} + \eta \cdot (N_{1,t} + \delta N_{2,t})$
  $t \leftarrow t + 1$
until convergence criteria met

Algorithm 2 Project (Zhao et al., 2024)

Require: Gradient matrix $G_t \in \mathbb{R}^{m\times n}$ with $m \leq n$, rank r, subspace change frequency T, and current step t
if $t \bmod T = 0$ **then**
  $U, S, V \leftarrow \mathrm{SVD}(G_t)$
  $P_t \leftarrow U[:, :r]$ {Initialize left projector as $m \leq n$}
else
  $P_t \leftarrow P_{t-1}$ {Reuse the previous projector}
end if
output $P_t^\top G_t$

4. Theory

In this section, we provide theoretical arguments to support the implementation of AdaMem.

Algorithm 3 Adafactor Preconditioner (Shazeer & Stern, 2018)

Operations such as squaring matrices and division of matrices are done elementwise.

Require: Momentum matrix $M_t \in \mathbb{R}^{r \times n}$, previous step moment estimates Z_{t-1}, decay rate $\beta_2 = .95$

$R_{t-1}, C_{t-1} \leftarrow Z_{t-1}$

$R_t = \beta_2 R_{t-1} + (1-\beta_2)(M_t^2)1_n$ {Update R_t, C_t as done in Adafactor}

$C_t = \beta_2 C_{t-1} + (1-\beta_2)1_r^\top (M_t^2)$

$\hat{V}_t = R_t C_t^T / (1_r^\top R_t)$ {Rank-1 approximation of second moment matrix, as done in Adafactor.}

output $\frac{M_t}{\sqrt{\hat{V}_t}}, Z_t$

Algorithm 4 One Sided Adafactor Preconditioner

Require: (Residual) gradient matrix $S_t \in \mathbb{R}^{m \times n}$, previous step moment estimates Z_{t-1}, decay rate $\beta_2 = .95$

$R_{t-1} \leftarrow Z_{t-1}$ {Only R_t is maintained and used for preconditioning.}

$R_t = \beta_2 R_{t-1} + (1-\beta_2)(S_t^2)1_n$

$Z_t \leftarrow R_t$

$\hat{V}_t = R_t 1_m^T / m$

output $\frac{S_t}{\sqrt{\hat{V}_t}}, Z_t$

4.1. Momentum only needed in the top eigenvectors of Hessian

Consider a quadratic loss landscape given by $\mathcal{L}(w) = w^T H w$, where H is a postive-definite matrix. Let the eigenspectrum of H be given by λ_i for $i \in [d]$ where $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_d$. Let κ denote the condition number of H given by λ_1/λ_d. Then it is well known (Nesterov, 1983) that gradient descent with momentum can achieve a convergence rate of $(\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$, which is a quadratic speed up of convergence rate of gradient descent, which is $(\kappa-1)/(\kappa+1)$. For maintaining momentum in the top-k eigenspace with power-law decay in the eigenspectrum, that is representative of deep learning tasks (Murray et al., 2023), the following result holds:

Theorem 4.1. *Consider a quadratic loss landscape given by $\mathcal{L}(w) = w^T H w$, where the eigenspectrum of H follows a power-law decay, i.e, $\lambda_i \propto i^\alpha$ for some $\alpha > 0$. For $k = d^{2/3}$, there exists learning rates η_1 and η_2, and momentum β, such that using learning rate η_1 and momentum β in the top-k eigenspace and learning rate η_2 with no momentum in the complement, leads to a convergence rate of $(\kappa^{1/3}-1)/(\kappa^{1/3}+1)$.*

Proof. For $k = d^{2/3}$, consider the top-k eigenspace denoted by S_k. For power-law decay, the condition number of S_k is given by $\frac{1}{d^{2\alpha/3}} = \kappa^{2/3}$. Thus, by the convergence rate of gradient descent with momentum, we can set learning rates η_1 and momentum β in this space to get a convergence rate of $(\kappa^{1/3}-1)/(\kappa^{1/3}+1)$.

Let the complement of S_k be denoted by C_k. The condition number of C_k is given by $\frac{1}{d^{\alpha/3}} = \kappa^{1/3}$. Thus, by the convergence rate of gradient descent, we can set learning rate η_2 in this space to get a convergence rate of $(\kappa^{1/3}-1)/(\kappa^{1/3}+1)$. □

Note that as $d \to \infty$, with $k = d^{2/3}$, $\frac{k}{d} \to 0$. Thus, in higher dimensions, we need to maintain momentum in a very small subspace to get the benefits of momentum. Although notice that this small subspace should correspond to the top eigenspace of the Hessian.

4.2. Connection to Shampoo, identifying top eigenvectors

Given Theorem 4.1 we know that momentum is only needed in the top eigenspace of the Hessian. This means we need to identify the top eigenspace. Various optimizers such as K-FAC and Shampoo approximate the Hessian by Kronecker products. We will be using the following claim from Gupta et al. (2018):

Claim 4.2. *Under Shampoo's Hessian approximation, eigenvectors of Hessian are of the form Vector(uv^T) where u and v are the eigenvectors of $\mathbb{E}[GG^T]$ and $\mathbb{E}[G^T G]$ respectively.*

Further approximating eigenvectors of $\mathbb{E}[GG^T]$ and $\mathbb{E}[G^T G]$ by a single sample of G, we see that the top singular vectors of G indeed are an estimate of the top eigenvectors of the Hessian. This shows that the low-rank subspace used in Algorithm 1 is a natural estimate of the top eigenspace of the Hessian. We hope that this connection between Shampoo and low rank training can be generally useful in further understanding the dynamics of low rank training.

5. Experiments

Models and Dataset. Starting from the OLMo codebase (Groeneveld et al., 2024), we train decoder-only transformer models with sequence length of 512 in two sizes: 210m and 550m. The models have widths of 1024, 1280, and depths of 12, 24. The batch sizes are $1024, 2048$. All models, except those using GaLore, are trained for 8000 steps with 2500 steps of warmup followed by cosine decay. For GaLore, we train for longer to take into account the fact that one gradient step of GaLore can be implemented with fewer FLOPs (see Appendix B). All of our models approximately match chinchilla (Hoffmann et al., 2022) style 20x scaling of tokens with respect to model size. The MLP hidden dimension is 4x the width. The activation function is GeLU

(Hendrycks & Gimpel, 2016). We use RoPE positional encodings (Su et al., 2024). Attention heads are always of dimension 64. We use PyTorch's default LayerNorm. Following Wortsman et al. (2024), we do not learn biases for the linear layers or LayerNorms. We train in mixed precision with bfloat16. We use the T5 tokenizer (Raffel et al., 2020) and train on the C4 dataset.

Optimizer Hyperparameter Sweeps. Following Touvron et al. (2023); Biderman et al. (2023), we fix $\beta_1 = .9, \beta_2 = .95$, and $\epsilon = 1e-8$ (though we study ablating β_2 in Section 6.1). We use weight decay of .01. For Adafactor and Adam, we sweep over the learning rates $[1.0e-4, 3.16e-3, 1.0e-3, 3.16e-3, 1.0e-2]$. For GaLore, we sweep over learning rates $[3.16e-4, 1.0e-3, 3.16e-3, 1.0e-2, 3.16e-2]$ and $[.125, .25, .5]$ for the α parameter. For AdaMeM, we sweep over learning rates $[3.16e-4, 1.0e-3, 3.16e-3, 1.0e-2]$ and $[.5, 1]$ for the δ parameter.

Results. In Figure 1, we compare our algorithm (AdaMeM) to Adafactor and Adam as baselines and the prior work of GaLore. In these plots we plot final validation loss as a function of rank used by AdaMeM and GaLore. We observe that while AdaMeM is always better than Adafactor, GaLore is only better than Adafactor at large ranks. In the GaLore paper, the low rank approximation is not applied to the first and last layer. We do the same in the right and middle figures by training the first and last layers with Adafactor for AdaMeM and Adam for GaLore. Here we see that while GaLore and AdaMeM perform nearly equally at the highest rank we considered (half of full rank), at lower ranks, AdaMeM performs significantly better. In the last plot, low rank approximation is applied to all layers, and we see that AdaMeM outperforms GaLore at all ranks considered.

6. Ablations

Other Design Choices. While designing AdaMeM, we tried some other variants which were less performant than AdaMeM. Variant 1 corresponds to directly adding the low rank momentum and the residual gradient and preconditioning in the original space by Adafactor, and Variant 2 is the same as AdaMeM except we use the standard Adafactor instead of one sided Adafactor for the residual gradient. We show their performance in Figure 2. We note that at smaller ranks, all of these variants outperform GaLore, which shows the importance of using full rank gradients.

Momentum and Batch Size. Prior works (Kidambi et al., 2018; Lee et al., 2022; Wang et al., 2024) have given empirical and theoretical evidence that momentum is only helpful at large batch sizes. In Figure 3, we confirm this for our setup where we vary batch size while keeping overall token

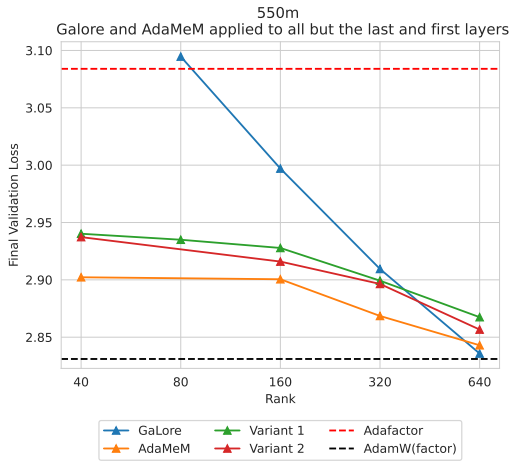

Figure 2. Performance comparison of variants of AdaMeM.

count approximately constant. This argues that for small batch sizes, Adafactor (without momentum) is as good as Adam. But for large batch sizes, momentum is needed, and as we have shown, that a large fraction of this gap can be recovered by using AdaMeM at a fraction of the memory cost of Adam or Adafactor with momentum.

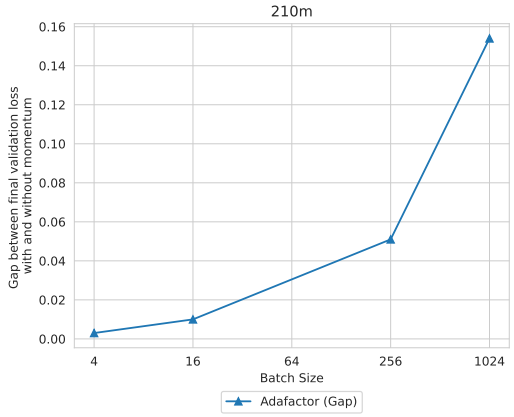

Figure 3. Effect of Momentum across batch sizes

6.1. Reproducing GaLore in their codebase; Ablating β_2

Algorithms	$\beta_2 = .95$	$\beta_2 = .999$
GaLore	3.205	3.23
Adam	3.136	3.22

Table 1. Performance of Adam and GaLore for different values of β_2. Results are for the 130M model used in the GaLore (Zhao et al., 2024) work.

The GaLore paper found that GaLore nearly matches Adam even with rank set to half of the full rank. We did not find this to be the case in Figure 1. We reproduce their

experiments in their codebase and show that this was due to non-optimal value of β_2. They used $\beta_2 = .999$, while the standard value used in language modelling (Biderman et al., 2023; Touvron et al., 2023; Wortsman et al., 2024) is $\beta_2 = .95$, which is what we have used for our experiments. Table 1 gives the performance of GaLore and Adam for $\beta_2 = .95$ and $\beta_2 = .999$. We see that

- $\beta_2 = .95$ outperforms $\beta_2 = .999$ for both algorithms.
- For $\beta_2 = .95$, Adam outperforms GaLore with rank set to half of the full rank.

Since we are reproducing these experiments from the GaLore (Zhao et al., 2024) work, we did not run GaLore for additional steps as described in Appendix B.

7. Limitations

The strongest limitation of our work and of other low rank optimizers such as ReLoRA and GaLore is that they are only practically useful to reduce memory in situations where momentum is helpful. This is because if momentum is not helpful then one can use Adafactor which has negligible memory overhead[2] above the weights themselves. In particular, the other memory usage is due to activations which grows with batch size and as prior works (as well our results in Section 6) have found, momentum is more beneficial with large batch sizes. In this work we have focused on optimizer memory usage, we leave the exploration of regimes in which optimizers such as ours can reduce *total memory usage* to future work.

8. Conclusion

In this work, we propose a memory efficient variant of Adafactor called AdaMem. It combines the preconditioning of Adafactor with low rank momentum ideas of works like LoRA and GaLore. We empirically show that this leads to significantly better performance for low-rank momentum as compared to GaLore, and also provide a theoretical justification for the method. Our method provides a specific recipe for combining preconditioning and momentum in a low-rank subspace, and further exploration within this space is a promising research direction.

Acknowledgments

SK and DM acknowledge support from the Office of Naval Research under award N00014-22-1-2377 and the National Science Foundation Grant under award #IIS 2229881. This work has been made possible in part by a gift from the Chan Zuckerberg Initiative Foundation to establish the Kempner Institute for the Study of Natural and Artificial Intelligence. NV and DM are supported by a Simons Investigator Fellowship, NSF grant DMS-2134157, DARPA grant W911NF2010021,and DOE grant DE-SC0022199.

[2]This holds under the AdaLomo implementation of Adafactor as discussed in Appendix A.

References

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., Skowron, A., Sutawika, L., and van der Wal, O. Pythia: A suite for analyzing large language models across training and scaling. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 2397–2430. PMLR, 2023. URL https://proceedings.mlr.press/v202/biderman23a.html.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C., Lu, Y., and Le, Q. V. Symbolic discovery of optimization algorithms. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *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-Confere
html.

Groeneveld, D., Beltagy, I., Walsh, P., Bhagia, A., Kinney, R., Tafjord, O., Jha, A. H., Ivison, H., Magnusson, I., Wang, Y., et al. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*, 2024.

Gupta, V., Koren, T., and Singer, Y. Shampoo: Preconditioned stochastic tensor optimization. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1842–1850. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/gupta18a.html.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W.,

Vinyals, O., and Sifre, L. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022. doi: 10.48550/ARXIV.2203.15556. URL https://doi.org/10.48550/arXiv.2203.15556.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. 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=nZeVKeeFYf9.

Kidambi, R., Netrapalli, P., Jain, P., and Kakade, S. M. On the insufficiency of existing momentum schemes for stochastic optimization. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJTutzbA-.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *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.

Lee, K., Cheng, A. N., Paquette, E., and Paquette, C. Trajectory of mini-batch momentum: Batch size saturation and convergence in high dimensions. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=z9poo2GhOh6.

Lialin, V., Shivagunde, N., Muckatira, S., and Rumshisky, A. Relora: High-rank training through low-rank updates. 2023.

Lv, K., Yan, H., Guo, Q., Lv, H., and Qiu, X. Adalomo: Low-memory optimization with adaptive learning rate. *CoRR*, abs/2310.10195, 2023a. doi: 10.48550/ARXIV.2310.10195. URL https://doi.org/10.48550/arXiv.2310.10195.

Lv, K., Yang, Y., Liu, T., Gao, Q., Guo, Q., and Qiu, X. Full parameter fine-tuning for large language models with limited resources. *CoRR*, abs/2306.09782, 2023b. doi: 10.48550/ARXIV.2306.09782. URL https://doi.org/10.48550/arXiv.2306.09782.

Murray, M., Jin, H., Bowman, B., and Montufar, G. Characterizing the spectrum of the NTK via a power series expansion. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Tvms8xrZHyR.

Nesterov, Y. A method of solving a convex programming problem with convergence rate o(1/k**2), 1983. URL https://cir.nii.ac.jp/crid/1370017280653524239.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, H. F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S. M., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X. L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J., Tsimpoukelli, M., Grigorev, N., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M. J., Hechtman, B. A., Weidinger, L., Gabriel, I., Isaac, W., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K., Stanway, J., Bennett, L., Hassabis, D., Kavukcuoglu, K., and Irving, G. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL https://arxiv.org/abs/2112.11446.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21 (140):1–67, 2020.

Robbins, H. and Monro, S. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951. doi: 10.1214/aoms/1177729586. URL https://doi.org/10.1214/aoms/1177729586.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In Dy, J. G. and Krause, A. (eds.), *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*, pp. 4603–4611. PMLR, 2018. URL http://proceedings.mlr.press/v80/shazeer18a.html.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation

language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/ARXIV.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Wang, R., Malladi, S., Wang, T., Lyu, K., and Li, Z. The marginal value of momentum for small learning rate SGD. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3JjJezzVkT.

Wortsman, M., Liu, P. J., Xiao, L., Everett, K. E., Alemi, A. A., Adlam, B., Co-Reyes, J. D., Gur, I., Kumar, A., Novak, R., Pennington, J., Sohl-Dickstein, J., Xu, K., Lee, J., Gilmer, J., and Kornblith, S. 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.

Zhai, X., Kolesnikov, A., Houlsby, N., and Beyer, L. Scaling vision transformers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pp. 1204–1213. IEEE, 2022. doi: 10.1109/CVPR52688.2022.01179. URL https://doi.org/10.1109/CVPR52688.2022.01179.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. 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.

A. LOMO Optimizer

LOMO optimizer Standard deep learning libraries such as PyTorch first compute the gradient for all parameters in the network and then apply it. This leads to a space requirement which is 2x that of storing just the weights of the network. LOMO (Lv et al., 2023b) instead is an implementation which applies the gradients as soon as they are computed and hence removes the 2x blowup in optimizer memory.

AdaLomo AdaLomo (Lv et al., 2023a) is an implementation of Adafactor which applies gradients of a layer as soon as they are computed, this removes the overhead of storing the gradients. With this implementation the space requirement of the optimizer is negligible as compared to that of the weights themselves.

B. Faster Implementation of GaLore

We note here that GaLore can be implemented faster than usual full rank algorithms. This is similar to the computational benefits of LoRA. To see this note that GaLore computes $P^T G$ for $P \in \mathbb{R}^{m \times r}, G \in \mathbb{R}^{m \times n}$. Let $A \in \mathbb{R}^{m \times z}$ denote the inputs to the layer in question where $z =$ sequence length $\times$ batch size. Similarly let $B \in \mathbb{R}^{n \times z}$ denote the output gradients. Then $G = AB^T$. Now rather than computing $P^T G$ as first computing $G = AB^T$ and then computing $P^T G$ we can instead first compute $C = P^T A$ and then compute CB^T. This changes the runtime of this part of backprop from $mn(z + r)$ to $(m + n)rz$. All of our GaLore runs take this speedup into account and run GaLore for longer to equalize FLOPs.