GaLore-mini: Low Rank Gradient Learning with Fewer Learning Rates

Anonymous Author(s)
Affiliation
Address
email

Abstract

Training large language models (LLMs) requires a significant memory capacity, mainly because of its vast number of model weights and optimizer states. In this work, we introduce a new memory-efficient optimizer called Galore-mini. Galore-mini exploits the inherent low-rank properties of weight gradients and the Hessian structure of transformers. These two characteristics significantly reduce the memory overhead by allowing the optimization states to be maintained in a low-rank format, while parameters are grouped into blocks, each sharing the same learning rate. However, directly combining these two strategies can easily lead to significant training instabilities. We explore several possible combinations and propose a strategy that stabilizes the training process. GaLore-mini reduces memory by 40% compared to GaLore and 81% compared to AdamW, while maintaining performance on par with vanilla AdamW. Experiments on LLaMA models ranging from 130M to 1B parameters demonstrate the effectiveness of GaLore-mini.

1 Introduction

In recent years, large foundation models have demonstrated outstanding performance across various domains, including natural language processing, speech recognition, and image generation. However, the immense number of parameters in these models has led to a substantial increase in the demand for computational resources during training. Among these, memory requirements have emerged as a critical bottleneck in the development of large foundation models. For instance, the popular Llama 7B model, with approximately 7 billion parameters, requires around 28GB of memory when stored in 32-bit full precision. When training from scratch using the Adam optimizer, which requires storing another two additional optimization states, m and v, for each parameter (doubling the memory usage), the total memory requirement for the optimizer reaches 56GB. This high memory demand significantly increases the cost of training large models, especially when the number of parameters further increases, the demand for memory increases rapidly, and the contradiction becomes more acute. Therefore, it is crucial to explore large model training solutions with low memory resources.

Recently, two studies, GaLore [1] and Adam-mini [2], have made initial progress in reducing optimizer memory usage. GaLore leverages the low-rank properties of parameter gradients, projecting the gradients into a low-rank space to update m and v, achieving exceptional memory efficiency. As an independent study of GaLore, Adam-mini explores the Hessian structure of Transformers and proposes to use the same v within each parameter block, where the parameter blocks are defined under certain principles. This approach can effectively eliminate most of v without any adverse effects, saving a lot of memory.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

Our Approach In this study, we found that the methods mentioned above are orthogonal. Therefore, we integrated the ideas of Adam-mini and GaLore to propose GaLore-mini. Our main contributions are as follows:

- GaLore-mini is a co-design solution that updates the optimizer in a low-rank space while simultaneously reducing the number of learning rates by degenerating v. A comparison of memory usage across various methods is shown in Figure 1. GaLore-mini reduces memory requirements by 40% compared to GaLore and by 81% compared to AdamW.
- Combining GaLore and Adam-mini was not a straightforward task. We explore three possible combinations and two of them encounter loss spikes. Fortunately, one of our trials work well and it effectively stabilize the training.
- In LLM experiments with parameters ranging from 125M to 1B, GaLore-mini performs comparably to AdamW.

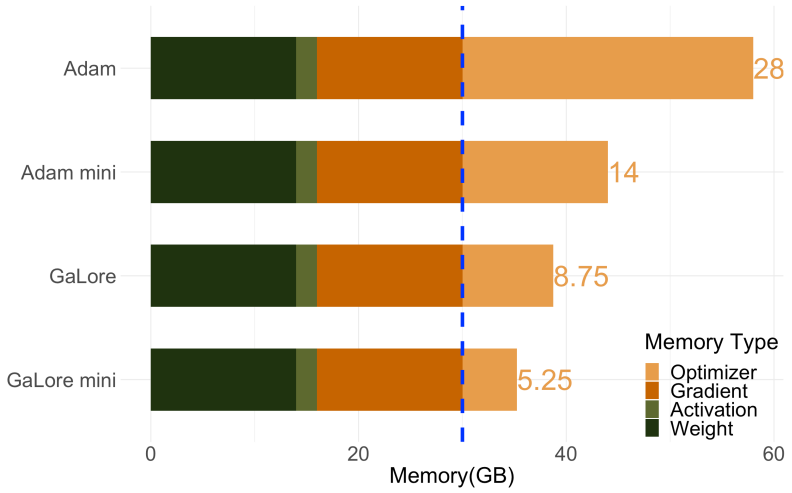


Figure 1: Memory consumption of different approaches for pre-training Llama-7B with FP16.

2 Related Works

Low-rank-based methods. To reduce memory demands, one approach is to decrease the number of parameters that need to be updated. LoRA [3], as a foundational work in this area, freezes the pre-trained weights and introduces additional trainable low-rank matrices, optimizing weights within a low-rank subspace. This approach significantly reduces resource consumption. However, LoRA is primarily used for model fine-tuning and cannot achieve performance comparable to full-rank fine-tuning. Subsequently, a variant known as ReLoRA [4] was introduced, which enables pre-training and achieves baseline performance; however, it still requires full-rank training as a warm-up.

Inspired by LoRA, [5] compresses gradients into a low-dimensional space through random projection, so that weights can be fully updated without excessive memory cost.

Based on prior research, GaLore [1] was proposed, whose key innovation is to utilize the natural low-rank structure of the gradient, rather than approximating the weight matrix itself as low-rank. GaLore is more memory-efficient in both pre-training and fine-tuning, performs well, and does not require full-rank warm-up like ReLoRA. Additionally, GaLore employs singular value decomposition (SVD) to compress gradients, offering more stability and accuracy compared to random projections. However, frequent SVD operations come with significant time overhead. In response to this challenge, Zhang et al. introduced Q-GaLore [6], which adaptively updates gradients based on statistical data from the training convergence process. Experiments have shown that Q-GaLore demonstrates highly competitive performance in both pre-training and fine-tuning tasks, while also having excellent memory efficiency.

Learning rate simplification methods. Another research approach focuses on reducing the number of learning rates without affecting model performance, thereby lowering memory requirements. Typical methods in this category include Adafactor and its variant CAME. Adafactor [7] and CAME [8] reduce memory consumption by applying low-rank matrix factorization to the second-order moment. Unfortunately, these methods are reported to exhibit worse performance compared to Adam.

Adam-mini [2] is a recent memory-efficient optimizer designed based on the Hessian structure of Transformers. Adam-mini cuts down memory by implying Adam's v: it proposes to use the same v within each parameter block, where the parameter blocks are defined under certain principles. Such a design can effectively eliminate $> 90\%$ Adam's v, and as a result, Adam-mini can save 45% to 50% memory of Adam.

3 GaLore-mini: Memory-Efficient Training

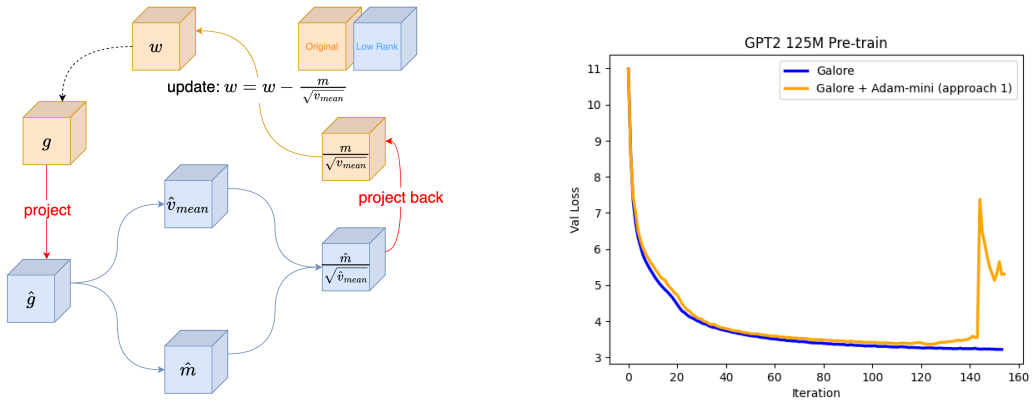
We now explore how to effectively combine GaLore and Adam-mini. We first summarize the key designs of both methods.

Key designs of GaLore: Firstly, GaLore projects the gradient g into a low dimensional space, and we call it $\hat{g}$. GaLore then computes Adam's 1st-order and 2nd-order momentum using $\hat{g}$, and we call them $\hat{m}$ and $\hat{v}$. Finally, GaLore projects the low-dimensional update matrix $\hat{m}/\sqrt{\hat{v}}$ back to the original space to update parameters.

Key designs of Adam-mini: Adam-mini partitions parameters into groups and assigns a shared 2nd-order momentum within each group. Such a 2nd-order momentum is calculated by taking the average of Adam's v within the group, and we denote it as v_{mean}.

To combine GaLore and Adam-mini, we need to decide how to calculate v_{mean} and how to apply it to update parameters. We find there are at least three possible approaches.

- **Approach 1:** Calculate v_{mean} using the low dimensional gradient $\hat{g}$, and then apply the learning rate $1/\sqrt{v_{mean}}$ to low dimensional $\hat{m}$. The procedure is illustrated in Figure 2 (a).
- **Approach 2:** Calculate v_{mean} using the original gradient g, and then apply the learning rate $1/\sqrt{v_{mean}}$ to low dimensional $\hat{m}$. The procedure is illustrated in Figure 3 (a).
- **Approach 3:** Calculate v_{mean} using the original gradient g, and then use the learning rate $1/\sqrt{v_{mean}}$ after the low dimensional $\hat{m}$ is projected back to the original space. The procedure is illustrated in Figure 4 (a).

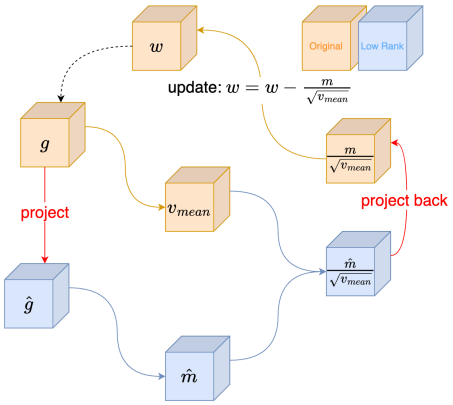


(a) Illustration of **Approach 1**

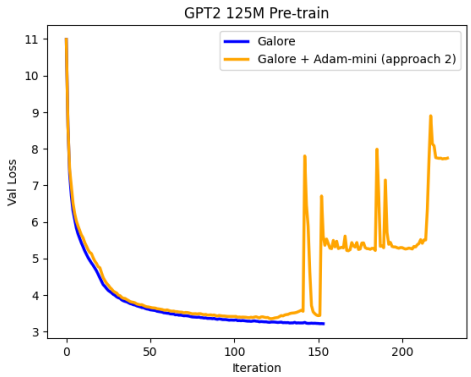
(b) Training curves of **Approach 1**

Figure 2: Pre-training results of the GPT-2 125M under **Approach 1**. We find that combining GaLore and Adam-mini under **Approach 1** will suffer training instability.

The training curves are shown in Figure 2, 3, and 4. We find that **Approach 1, 2** both encounter training instability. In contrast, **Approach 3** works well. One possible reason for the training instability is that the Hessian structure is changed in the low-rank space, so the original parameter partition in Adam-mini no long applies. In the following, we will adopt **Approach 3** and call it GaLore-mini.

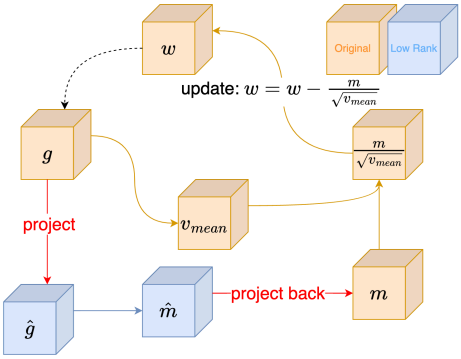


(a) Illustration of **Approach 2**

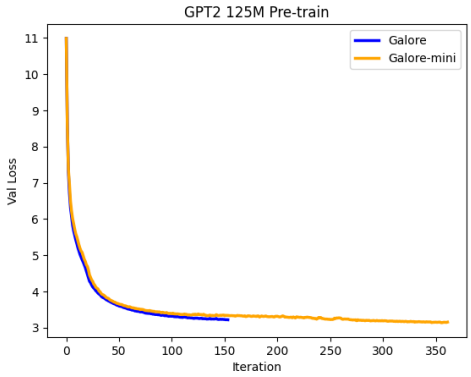


(b) Training curves of **Approach 2**

Figure 3: Pre-training results of the GPT-2 125M under **Approach 2**. We find that combining GaLore and Adam-mini under **Approach 2** will suffer training instability.



(a) Illustration of **Approach 3**



(b) Training curves of **Approach 3**

Figure 4: Pre-training results of the GPT-2 125M under **Approach 3**. We find that combining GaLore and Adam-mini under **Approach 3** works well. We call the resulting method GaLore-mini.

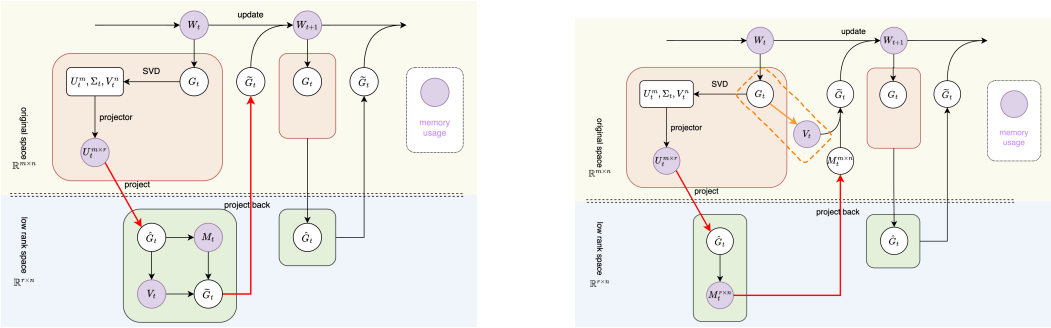
3.1 Detailed Form of GaLore-mini

Based on the above discussion, we now formally present the details of GaLore-mini here. GaLore-mini consists of two main components: initialization and iteration.

Initialization In the initialization phase ($t = 0$), we start with the weight matrix $W_0 \in \mathbb{R}^{m\times n}, m \leq n$ and set the following hyperparameters: low rank r, decay rates β_1 and β_2, first-order moment matrix M_0, second-order moment matrix V_0, learning rate η, scale factor α, low rank space change frequency T and the neural-wise parameter partition mentioned earlier.

Iteration During the iteration phase, we update the weight matrix step by step according to the process illustrated in Figure 5b, until the convergence condition is met. Specifically:

- Step 1: Compute the gradient matrix based on loss function: $G_t = \nabla_W L(W_t)$
- Step 2: Following the neural-wise partition, the second-order moment matrix V_t is updated (as indicated by the red dashed box in the figure) with bias correction applied;
- Step 3: According to the suggestion of Q-GaLore [6], the singular value decomposition of the gradient matrix is dynamically adjusted: $G_t = U_t^m \Sigma_t^{m\times n} (V_t^n)^{\mathrm{T}}$ and the projection



(a) GaLore (b) GaLore-mini

Figure 5: Comparison of the GaLore and GaLore-mini iteration processes. W_t represents the weight matrix, G_t denotes the gradient matrix, $U_t^{m\times r}$ is the projection matrix truncated to r columns, $\hat{G}_t$ is the gradient matrix projected into a low-rank space, and M_t, V_t represent the first and second order moment of the optimizer, respectively. Additionally, $\tilde{G}_t = \frac{M_t}{\sqrt{V_t}+\varepsilon\cdot\mathbf{I}^{m\times n}}$ is the gradient matrix used for updating the weights, and t indicates the time step.

matrix $U_t^{m\times r}$ is obtained by truncating U_t^m by column according to the hyperparameter r. Besides this, the projection matrix from the previous moment is reused: $U_t^{m\times r} = U_{t-1}^{m\times r}$;

- Step 4: The gradient matrix is projected as follows: $\hat{G}_t = (U_t^{m\times r})^{\mathrm{T}} G_t$
- Step 5: In the low-rank space, the first-order moment matrix is updated as $M_t^{r\times n} = \beta_1 M_{t-1}^{r\times n} + (1-\beta_1)\hat{G}_t$, with bias correction applied;
- Step 6: The first-order moment matrix in the low-rank space is mapped back to the original space ($M_t^{m\times n}$) and combined with the second-order moment matrix V_t to update the adaptive learning rate as follows: $\eta\widetilde{G}_t = \eta\frac{M_t}{\sqrt{V_t}+\varepsilon\cdot\mathbf{I}^{m\times n}}$;
- Step 7: The weight matrix is updated: $W_t = W_{t-1} + \eta\widetilde{G}_t$

Remark 1 *We compute V_t in the original space, while calculating $M_t^{r\times n}$ in the low-rank space. This design stems from our intent to leverage the well-established experience of Adam-mini as much as possible. Previously, we attempted to degenerate V_t in the low-rank space, as shown on the left side of Figure 5a, but this approach led to training instability. The details of this issue will be discussed further in Section 5.*

In summary, the algorithm for GaLore-mini is presented in Algorithm 1.

3.2 Analysis of Memory Consumption

Based on the analysis presented above, we can summarize the memory consumption as shown in Table 1 (Assume that -mini methods can be approximately regarded as eliminating all v values).

Table 1: Memory comparison of related methods

	GaLore-mini	GaLore	LoRA	Adam-mini	AdamW
Weights	mn	mn	$mn+mr+nr$	mn	mn
Optimizer	$mr+nr$	$mr+2nr$	$2mr+2nr$	mn	$2mn$

Taking one layer in the Llama architecture as an example, the layer is divided into two parts: self-attention and MLP. The main model parameters for self-attention include the weight matrices for Q, K, V: $W_Q, W_K, W_V \in \mathbb{R}^{h\times h}$, and the output weight matrix $W_O \in \mathbb{R}^{h\times h}$. The MLP consists of two linear layers: the first linear layer has a matrix of size $\mathbb{R}^{h\times 4h}$, and the second linear layer has a matrix of size $\mathbb{R}^{4h\times h}$.

Algorithm 1 GaLore-mini

Input: weight matrix $W_0 \in \mathbb{R}^{m\times n}$, low rank r, decay rates β_1 and β_2, first-order moment matrix M_0, second-order moment matrix V_0, learning rate η, scale factor α, subspace change frequency T and the neural-wise parameter partition principle.

repeat

$G_t = \nabla_W L(W_t) \in \mathbb{R}^{m\times n}$

Following the neural-wise partition principle, update $V_t = \beta_2 V_{t-1} + (1-\beta_2)\hat{G}_t$ with $V_t = \frac{V_t}{1-\beta_2^t}$

if t mod T = 0 **then**

SVD: $G_t = U_t^m \Sigma_t^{m\times n} (V_t^n)^{\mathrm{T}}$

$U_t^{m\times r}$, assuming $m \leq n$ {truncate by hyperparameter r to get projection matrix}

Adaptively update T based on the cosine similarity between adjacent projection matrices

else

$U_t^{m\times r} = U_{t-1}^{m\times r}$ {Reuse the previous projection matrix}

end if

Update in low rank space $\mathbb{R}^{r\times n}$

$\hat{G}_t = (U_t^{m\times r})^{\mathrm{T}} G_t$ {Project gradient into low rank space}

update $M_t^{r\times n} = \beta_1 M_{t-1}^{r\times n} + (1-\beta_1)\hat{G}_t$ with $M_t^{r\times n} = \frac{M_t^{r\times n}}{1-\beta_1^t}$

$M_t = U_t^{m\times r} M_t^{r\times n}$ {Project back to original space $\mathbb{R}^{m\times n}$}

$\widetilde{G}_t = \alpha \frac{M_t}{\sqrt{V_t}+\varepsilon\cdot \mathbf{I}^{m\times n}}$

$W_t = W_{t-1} + \eta \widetilde{G}_t$

$t = t+1$

until convergence criteria met

return W_t

Thus, we can calculate the memory consumption of the optimizer for each method as follows: AdamW ($24h^2$), Adam-mini ($12h^2$), GaLore ($30hr$), and GaLore-mini ($18hr$). It is evident that GaLore-mini saves 40% of memory compared to GaLore, and 81.25% compared to AdamW (when $h = 4096$ and $r = 1024$).

4 Experiments

Implementation Details We evaluate the effectiveness of GaLore-mini on C4 pre-training tasks using LLaMA models with parameter sizes of 130M, 350M, and 1B. The models are trained on 2.2B, 6.4B, and 2.4B tokens, respectively. Other hyperparameters follow the original setup[1].

Baselines We compare GaLore-mini against five baselines: (i) `Full`: the original training approach using the Adam optimizer; (ii) `Low-Rank`: where the original weights are decomposed into two low-rank components $W = AB$, and the low-rank weights are optimized using the Adam optimizer; (iii) `LoRA` [3], which introduces a low-rank adapter, optimizing only the adapter with $W = W_0 + AB$ while keep W_0 freezed; (iv) `ReLoRA` [4]: an extension of LoRA, where the low-rank adapter is periodically merged back into the weight W_o and re-initialized for the next training phase; and (v) `GaLore` [1]: which maintains low-rank optimization states while keeping full-rank weights and trains using the Adam optimizer.

End-to-End Results As shown in Table 2, we report the perplexity on the validation set and the estimated memory consumption of the weights and optimization states. Compared to other baselines, GaLore-mini requires significantly less memory overhead while maintaining perplexity comparable to the `Full` training approach. For instance, in the 1B parameter training, GaLore-mini uses less than half (48.59%) of the memory required by `Full` with no performance loss (18.73 v.s. 18.67).

5 Discussion and Future Work

We proposed GaLore-mini, a memory-efficient optimizer that reduces memory by 40% compared to GaLore and 81% compared to AdamW. Experiments on LLaMA models ranging from 130M to 1B demonstrate that GaLore-mini can perform on par with vanilla AdamW.

Table 2: Pre-traing on C4 dataset with LLaMA models. The model size ranges from 130M to 1B and the reported memory counts for the weight and optimization states.

Methods	130M		350M		1B	
	Perplexity	Memory	Perplexity	Memory	Perplexity	Memory
Full	25.08	0.76G	18.80	2.06G	18.73	7.80G
Low-Rank	45.51	0.54G	37.41	1.08G	144.03	3.57G
LoRA	33.92	0.80G	25.58	1.76G	29.08	6.17G
ReLoRA	29.37	0.80G	29.08	1.76G	31.82	6.17G
GaLore	25.36	0.52G	18.95	1.22G	19.29	4.38G
GaLore-mini	25.59	0.44G	19.89	1.04G	18.67	3.79G

References

[1] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

[2] Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. *arXiv preprint arXiv:2406.16793*, 2024.

[3] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[4] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *The Twelfth International Conference on Learning Representations*, 2023.

[5] Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. *arXiv preprint arXiv:2402.03293*, 2024.

[6] Zhenyu Zhang, Ajay Jaiswal, Lu Yin, Shiwei Liu, Jiawei Zhao, Yuandong Tian, and Zhangyang Wang. Q-galore: Quantized galore with int4 projection and layer-adaptive low-rank gradients. *arXiv preprint arXiv:2407.08296*, 2024.

[7] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[8] Yang Luo, Xiaozhe Ren, Zangwei Zheng, Zhuo Jiang, Xin Jiang, and Yang You. Came: Confidence-guided adaptive memory efficient optimization. *arXiv preprint arXiv:2307.02047*, 2023.