GALORE: MEMORY-EFFICIENT LLM TRAINING BY GRADIENT LOW-RANK PROJECTION

Jiawei Zhao[1] Zhenyu Zhang[3] Beidi Chen[2,4] Zhangyang Wang[3]
Anima Anandkumar[1,*] Yuandong Tian[2,*]
[1] California Institute of Technology
[2] Meta AI
[3] University of Texas at Austin
[4] Carnegie Mellon University
[*] Equal advising
{jiawei,anima}@caltech.edu,{zhenyu.zhang,atlaswang}@utexas.edu,
beidic@andrew.cmu.edu & yuandong@meta.com

ABSTRACT

Training Large Language Models (LLMs) presents significant memory challenges, predominantly due to the growing size of weights and optimizer states. Common memory-reduction approaches, such as low-rank adaptation (LoRA), add a trainable low-rank matrix to the frozen pre-trained weight in each layer, reducing trainable parameters and optimizer states. However, such approaches typically underperform training with full-rank weights in both pre-training and fine-tuning stages since they limit the parameter search to a low-rank subspace and alter the training dynamics, and further, may require full-rank warm start. In this work, we propose Gradient Low-Rank Projection (**GaLore**), a training strategy that allows *full-parameter* learning but is more *memory-efficient* than common low-rank adaptation methods such as LoRA. Our approach reduces memory usage by up to 65.5% in optimizer states while maintaining both efficiency and performance for pre-training on LLaMA 1B and 7B architectures with C4 dataset with up to 19.7B tokens, and on fine-tuning RoBERTa on GLUE tasks. Our 8-bit GaLore further reduces optimizer memory by up to 82.5% and total training memory by 63.3%, compared to a BF16 baseline. Notably, we demonstrate, for the first time, the feasibility of pre-training a 7B model on consumer GPUs with 24GB memory (e.g., NVIDIA RTX 4090) without model parallel, checkpointing, or offloading strategies.

1 INTRODUCTION

Large Language models (LLMs) have shown impressive performance across multiple disciplines, including conversational AI and language translation. However, pre-training and fine-tuning LLMs require not only a huge amount of computation but is also memory intensive. The memory requirements include not only billions of trainable parameters, but also their gradients and optimizer states (e.g., gradient momentum and variance in Adam) that can be larger than parameter storage themselves (Raffel et al., 2023; Touvron et al., 2023; Chowdhery et al., 2022). For example, pre-training a LLaMA 7B model from scratch with a single batch size requires at least 58 GB memory (14GB for trainable parameters, 42GB for Adam optimizer states and weight gradients, and 2GB

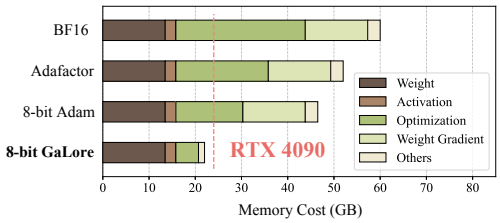

Figure 1: Memory consumption of pre-training a LLaMA 7B model with a token batch size of 256 on a single device, without activation checkpointing and memory offloading.

for activations[1]). This makes the training not feasible on consumer-level GPUs such as NVIDIA RTX 4090 with 24GB memory.

In addition to engineering and system efforts, such as gradient checkpointing Chen et al. (2016), memory offloading Rajbhandari et al. (2020), etc., to achieve faster and more efficient distributed training, researchers also seek to develop various optimization techniques to reduce the memory usage during pre-training and fine-tuning.

Parameter-efficient fine-tuning (PEFT) techniques allow for the efficient adaptation of pre-trained language models (PLMs) to different downstream applications without the need to fine-tune all of the model's parameters (Ding et al., 2022). Among them, the popular Low-Rank Adaptation (LoRA Hu et al. (2021)) *reparameterizes* weight matrix $W \in \mathbb{R}^{m \times n}$ into $W = W_0 + BA$, where W_0 is a frozen full-rank matrix and $B \in \mathbb{R}^{m \times r}$, $A \in \mathbb{R}^{r \times n}$ are additive low-rank adaptors to be learned. Since the rank $r \ll \min(m, n)$, A and B contain fewer number of trainable parameters and thus smaller optimizer states. LoRA has been used extensively to reduce memory usage for fine-tuning in which W_0 is the frozen pre-trained weight. Its variant ReLoRA is also used in pre-training, by periodically updating W_0 using previously learned low-rank adaptors (Lialin et al., 2023).

However, many recent works demonstrate the limitation of such a low-rank reparameterization. For fine-tuning, LoRA is not shown to reach a comparable performance as full-rank fine-tuning Xia et al. (2024). For pre-training from scratch, it is shown to require a full-rank model training as a warmup (Lialin et al., 2023), before optimizing in the low-rank subspace. There are two possible reasons: (1) the optimal weight matrices may not be low-rank, and (2) the reparameterization changes the gradient training dynamics.

Algorithm 1: GaLore, PyTorch-like

```
for weight in model.parameters():
   grad = weight.grad
   # original space -> compact space
   lor_grad = project(grad)
   # update by Adam, Adafactor, etc.
   lor_update = update(lor_grad)
   # compact space -> original space
   update = project_back(lor_update)
   weight.data += update
```

Our approach: To address the above challenge, we propose Gradient Low-rank Projection (**GaLore**), a training strategy that allows *full-parameter* learning but is more *memory-efficient* than common low-rank adaptation methods, such as LoRA. Our key idea is to leverage the slow-changing low-rank structure of the *gradient* $G \in \mathbb{R}^{m \times n}$ of the weight matrix W, rather than trying to approximate the weight matrix itself as low rank.

Motivated by the fact that the gradient matrix G becomes low-rank during training. Then, we propose GaLore that computes two projection matrices $P \in \mathbb{R}^{m \times r}$ and $Q \in \mathbb{R}^{n \times r}$ to project the gradient matrix G into a low-rank form $P^\top GQ$. In this case, the memory cost of optimizer states, which rely on component-wise gradient statistics, can be substantially reduced. Occasional updates of P and Q (e.g., every 200 iterations) incur minimal amortized additional computational cost. GaLore is more memory-efficient than LoRA as shown in Table 1. In practice, this yields up to 30% memory reduction compared to LoRA during pre-training.

We demonstrate that GaLore works well in both LLM pre-training and fine-tuning. When pre-training LLaMA 7B on C4 dataset, 8-bit GaLore, combined with 8-bit optimizers and layer-wise weight updates techniques, achieves comparable performance to its full-rank counterpart, with less than 10% memory cost of optimizer states.

Notably, for pre-training, GaLore keeps low memory throughout the entire training, without requiring full-rank training warmup like ReLoRA. Thanks to GaLore's memory efficiency, for the first time it is possible to train LLaMA 7B from scratch on a single GPU with 24GB memory (e.g., on NVIDIA RTX 4090), without any costly memory offloading techniques (Fig. 1).

GaLore is also used to fine-tune pre-trained LLMs on GLUE benchmarks with comparable or better results than existing low-rank methods. When fine-tuning RoBERTa-Base on GLUE tasks with a rank of 4, GaLore achieves an average score of 85.89, outperforming LoRA, which achieves a score of 85.61.

[1]The calculation is based on LLaMA architecture, BF16 numerical format, and maximum sequence length of 2048.

As a gradient projection method, GaLore is independent of the choice of optimizers and can be easily plugged into existing ones with only two lines of code, as shown in Algorithm 1. In addition, its performance is insensitive to very few hyper-parameters it introduces.

2 GALORE: GRADIENT LOW-RANK PROJECTION

Regular full-rank training. At time step t, $G_t = -\nabla_W \varphi_t(W_t) \in \mathbb{R}^{m\times n}$ is the backpropagated (negative) gradient matrix. Then the regular pre-training weight update can be written down as follows (η is the learning rate):

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t = W_0 + \eta \sum_{t=0}^{T-1} \rho_t(G_t) \tag{1}$$

where $\tilde{G}_t$ is the final processed gradient to be added to the weight matrix and ρ_t is an entry-wise stateful gradient regularizer (e.g., Adam). The state of ρ_t can be memory-intensive. For example, for Adam, we need $M, V \in \mathbb{R}^{m\times n}$ to regularize the gradient G_t into $\tilde{G}_t$:

$$M_t = \beta_1 M_{t-1} + (1-\beta_1) G_t \tag{2}$$

$$V_t = \beta_2 V_{t-1} + (1-\beta_2) G_t^2 \tag{3}$$

$$\tilde{G}_t = M_t / \sqrt{V_t + \epsilon} \tag{4}$$

Here G_t^2 and $M_t/\sqrt{V_t+\epsilon}$ means element-wise multiplication and division. η is the learning rate. Together with $W \in \mathbb{R}^{m\times n}$, this takes $3mn$ memory.

Gradient Low-Rank Projection. Since G_t may have a low-rank structure, if we can keep the gradient statistics of a small "core" of gradient G_t in optimizer states, rather than G itself, then the memory consumption can be reduced substantially. This leads to our proposed GaLore strategy:

Definition 2.1 (Gradient Low-rank Projection (**GaLore**))**.** Gradient low-rank projection (**GaLore**) denotes the following gradient update rules:

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t, \qquad \tilde{G}_t = P_t \rho_t(P_t^\top G_t Q_t) Q_t^\top, \tag{5}$$

where $P_t \in \mathbb{R}^{m\times r}$ and $Q_t \in \mathbb{R}^{n\times r}$ are projection matrices.

Setting P and Q. It is straightforward that P and Q should project into the subspaces corresponding to the first few largest eigenvectors in G for faster convergence. We find these largest eigenvectors by performing Singular Value Decomposition (SVD) on G_t:

$$G_t = USV^\top \approx \sum_{i=1}^{r} s_i u_i v_i^\top \tag{6}$$

$$P_t = [u_1, u_2, ..., u_r], \quad Q_t = [v_1, v_2, ..., v_r] \tag{7}$$

Composition of low-rank subspaces. For a complex optimization problem such as LLM pre-training, it may be difficult to capture the entire gradient trajectory with a single low-rank subspace determined by a fixed P and Q. One reason is that the principal subspaces may change over time. In fact, if we keep the same projection P and Q, then the learned weights will only grow along these subspaces, which is not longer full-parameter training. Fortunately, for this, GaLore can switch subspaces during training and learn full-rank weights without increasing the memory footprint.

We allow GaLore to switch across low-rank subspaces:

$$W_t = W_0 + \Delta W_{T_1} + \Delta W_{T_2} + \ldots + \Delta W_{T_n}, \tag{8}$$

where $t \in \left[\sum_{i=1}^{n-1} T_i, \sum_{i=1}^{n} T_i\right]$ and $\Delta W_{T_i} = \eta \sum_{t=0}^{T_i-1} \tilde{G}_t$ is the summation of all T_i updates within the i-th subspace. When switching to i-th subspace at step $t = T_i$, we re-initialize the projector P_t and Q_t by performing SVD on the current gradient G_t by Equation 6.

Table 2: Comparison with low-rank algorithms on pre-training various sizes of LLaMA models on C4 dataset. Validation perplexity is reported, along with a memory estimate of the total of parameters and optimizer states based on BF16 format. The actual memory footprint of GaLore is reported in Fig. 1 and Fig. 2.

	60M	**130M**	**350M**	**1B**
Full-Rank	34.06 (0.36G)	25.08 (0.76G)	18.80 (2.06G)	15.03 (7.80G)
GaLore	**34.88** (0.24G)	**25.36** (0.52G)	**18.95** (1.22G)	**14.57** (5.78G)
Low-Rank	78.18 (0.26G)	45.51 (0.54G)	37.41 (1.08G)	135.23 (6.37G)
LoRA	34.99 (0.36G)	33.92 (0.80G)	25.58 (1.76G)	18.92 (8.96G)
ReLoRA	37.04 (0.36G)	29.37 (0.80G)	29.08 (1.76G)	17.64 (8.96G)
r/d_{model}	128 / 256	256 / 768	256 / 1024	1024 / 2048
Training Tokens	1.1B	2.2B	6.4B	13.1B

Reducing memory footprint of gradient statistics. GaLore significantly reduces the memory cost of optimizer that heavily rely on component-wise gradient statistics, such as Adam (Kingma & Ba, 2014). When $\rho_t \equiv \mathrm{Adam}$, by projecting G_t into its low-rank form $R_t = P_t^\top G_t Q_t$, Adam's gradient regularizer $\rho_t(R_t)$ only needs to track low-rank gradient statistics. where M_t and V_t are the first-order and second-order momentum, respectively. GaLore requires less memory than LoRA during training. As GaLore can always merge ΔW_t to W_0 during weight updates, it does not need to store a separate low-rank factorization BA. In total, GaLore requires $(mn + mr + 2nr)$ memory, while LoRA requires $(mn + 3mr + 3nr)$ memory. A comparison between GaLore and LoRA is shown in Table 1.

Per-layer weight updates. In practice, the optimizer typically performs a single weight update for all layers after backpropagation. This is done by storing the entire weight gradients in memory. To further reduce the memory footprint during training, we adopt *per-layer weight updates* technique to GaLore, which performs the weight update during backpropagation (Lv et al., 2023).

Table 1: Comparison between GaLore and LoRA. Assume $W \in \mathbb{R}^{m\times n}$ ($m \leq n$), rank r.

	GaLore	LoRA
Weights	mn	$mn + mr + nr$
Optim States	$mr + 2nr$	$2mr + 2nr$
Multi-Subspace	✓	✗
Pre-Training	✓	✗
Fine-Tuning	✓	✓

3 Experiments

We evaluate GaLore on both pre-training and
fine-tuning of LLMs. To evaluate its performance, we apply GaLore to train LLaMA-based large language models on the C4 dataset. C4 dataset is a colossal, cleaned version of Common Crawl's web crawl corpus, which is mainly intended to pre-train language models and word representations (Raffel et al., 2023). To best simulate the practical pre-training scenario, we train without data repetition over a sufficiently large amount of data, across a range of model sizes up to 7 Billion parameters. We also evaluate GaLore on fine-tuning LLMs on GLUE tasks to demonstrate its memory-efficient fine-tuning capability.

Comparison with low-rank methods We first compare GaLore with existing low-rank methods using Adam optimizer across a range of model sizes. **Full-Rank**: our baseline method that applies Adam optimizer with full-rank weights and optimizer states. **Low-Rank**: we also evaluate a traditional low-rank approach that represents the weights by learnable low-rank factorization: $W = BA$ (Kamalakara et al., 2022). **LoRA**: Hu et al. (2021) proposed LoRA to fine-tune pre-trained models with low-rank adaptors: $W = W_0 + BA$, where W_0 is fixed initial weights and BA is a learnable low-rank adaptor. **ReLoRA**: Lialin et al. (2023) is a variant of LoRA designed for pre-training, which periodically merges BA into W, and initializes new BA with a reset on optimizer states and learning rate.

For GaLore, we set subspace frequency T to 200 and scale factor α to 0.25 across all model sizes in Table 2. For each model size, we pick the same rank r for all low-rank methods, and we apply them to all multi-head attention layers and feed-forward layers in the models. We train all models using

Table 3: Evaluating GaLore for memory-efficient fine-tuning on GLUE benchmark using pre-trained RoBERTa-Base. We report the average score of all tasks.

	CoLA	STS-B	MRPC	RTE	SST2	MNLI	QNLI	QQP	Avg
Full Fine-Tuning	62.24	90.92	91.30	79.42	94.57	87.18	92.33	92.28	86.28
GaLore (rank=4)	60.35	**90.73**	**92.25**	**79.42**	**94.04**	**87.00**	**92.24**	91.06	**85.89**
LoRA (rank=4)	**61.38**	90.57	91.07	78.70	92.89	86.82	92.18	**91.29**	85.61
GaLore (rank=8)	60.06	**90.82**	**92.01**	**79.78**	**94.38**	**87.17**	92.20	91.11	**85.94**
LoRA (rank=8)	**61.83**	90.80	91.90	79.06	93.46	86.94	**92.25**	**91.22**	85.93

Adam optimizer with the default hyperparameters (e.g., $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$). We also estimate the memory usage based on BF16 format, including the memory for weight parameters and optimizer states. As shown in Table 2, GaLore outperforms other low-rank methods and achieves comparable performance to full-rank training. It even outperforms full-rank training on 1B model size with a validation perplexity of 14.57. Compared to LoRA and ReLoRA, GaLore requires less memory for storing model parameters and optimizer states. A detailed training setting of each model and our memory estimation for each method are provided in the appendix.

Scaling up to LLaMA 7B architecture. Scaling ability to 7B models is a key factor for demonstrating if GaLore is effective for practical LLM pre-training scenarios. We evaluate GaLore on an LLaMA 7B architecture with an embedding size of 4096 and total layers of 32. We train the model for 150K steps, using 8-node training in parallel with a total of 64 A100 GPUs. Due to computational constraints, we only compare 8-bit GaLore ($r = 1024$) with 8-bit Adam with a single trial without tuning the hyperparameters. After 150K steps, 8-bit GaLore achieves a perplexity of 14.65, which is comparable to 8-bit Adam with a perplexity of 14.48.

Memory-efficient fine-tuning GaLore not only achieves memory-efficient pre-training but also can be used for memory-efficient fine-tuning. We fine-tune pre-trained RoBERTa models on GLUE tasks using GaLore and compare its performance with a full fine-tuning baseline and LoRA. As shown in Table 3, GaLore achieves better performance than LoRA on most tasks with less memory footprint. This demonstrates that GaLore can serve as a full-stack memory-efficient training strategy for both LLM pre-training and fine-tuning.

Training 7B models on consumer GPUs with 24G memory. We measure the actual memory footprint of training LLaMA 7B models by various methods, with a token batch size of 256. As shown in Fig. 2, 8-bit GaLore requires significantly less memory than BF16 baseline and 8-bit Adam, and only requires 22.0G memory to pre-train LLaMA 7B with a small per-GPU token batch size (up to 500 tokens). This memory footprint is within 24GB VRAM capacity of a single GPU such as NVIDIA RTX 4090. While the batch size is small per GPU, it can be scaled up with data parallelism, which requires much lower bandwidth for inter-GPU communication, compared to model parallelism. Therefore, it is possible that GaLore can be used for elastic training Lin et al. 7B models on consumer GPUs such as RTX 4090s. Specifically, we present the memory breakdown in Fig. 1. It shows that 8-bit GaLore reduces 37.92G (63.3%) and 24.5G (52.3%) total memory compared to BF16 Adam baseline and 8-bit Adam, respectively. Compared to 8-bit Adam, 8-bit GaLore mainly reduces the memory in two parts: (1) low-rank gradient projection reduces 9.6G (65.5%) memory of storing optimizer states, and (2) using per-layer weight updates reduces 13.5G memory of storing weight gradients.

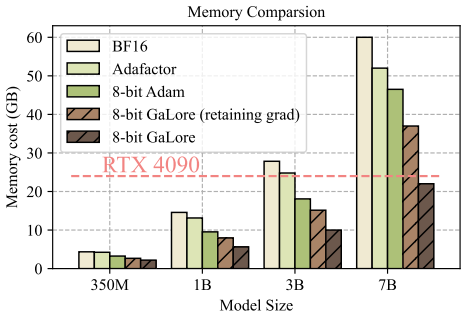

Figure 2: Memory usage for different methods at various model sizes, evaluated with a token batch size of 256. 8-bit GaLore (retaining grad) disables per-layer weight updates but stores weight gradients during training.

4 CONCLUSION

We propose GaLore, a memory-efficient pre-training and fine-tuning strategy for large language models. GaLore significantly reduces memory usage by up to 65.5% in optimizer states while maintaining both efficiency and performance for large-scale LLM pre-training and fine-tuning. We hope that our work will inspire future research on memory-efficient LLM training strategies from the perspective of low-rank gradient projection. We believe that GaLore will be a valuable tool for the community to train large language models with consumer-grade hardware and limited resources.

REFERENCES

Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory Efficient Adaptive Optimization.

Arslan Chaudhry, Naeemullah Khan, Puneet Dokania, and Philip Torr. Continual Learning in Low-rank Orthogonal Subspaces. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9900–9911. Curran Associates, Inc., 2020.

Han Chen, Garvesh Raskutti, and Ming Yuan. Non-Convex Projected Gradient Descent for Generalized Low-Rank Tensor Regression. *Journal of Machine Learning Research*, 20(5):1–37, 2019. ISSN 1533-7928.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost, April 2016.

Yudong Chen and Martin J. Wainwright. Fast low-rank estimation by projected gradient descent: General statistical and algorithmic guarantees, September 2015.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways, October 2022.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit Optimizers via Block-wise Quantization. *arXiv:2110.02861 [cs]*, October 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient Finetuning of Quantized LLMs, May 2023.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. Delta Tuning: A Comprehensive Study of Parameter Efficient Methods for Pre-trained Language Models, March 2022.

Guy Gur-Ari, Daniel A. Roberts, and Ethan Dyer. Gradient Descent Happens in a Tiny Subspace, December 2018.

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, October 2021.

Siddhartha Rao Kamalakara, Acyr Locatelli, Bharat Venkitesh, Jimmy Ba, Yarin Gal, and Aidan N. Gomez. Exploring Low Rank Training of Deep Neural Networks, September 2022.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014.

Brett W. Larsen, Stanislav Fort, Nic Becker, and Surya Ganguli. How many degrees of freedom do we need to train deep networks: A loss landscape perspective, February 2022.

Yoonho Lee and Seungjin Choi. Gradient-Based Meta-Learning with Learned Layerwise Metric and Subspace, June 2018.

Bingrui Li, Jianfei Chen, and Jun Zhu. Memory Efficient Optimizers with 4-bit States. https://arxiv.org/abs/2309.01507v3, September 2023.

Vladislav Lialin, Namrata Shivagunde, Sherin Muckatira, and Anna Rumshisky. ReLoRA: High-Rank Training Through Low-Rank Updates, December 2023.

Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. URL `http://arxiv.org/abs/1904.12043`.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full Parameter Fine-tuning for Large Language Models with Limited Resources, June 2023.

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, September 2023.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, May 2020.

Adithya Renduchintala, Tugrul Konuk, and Oleksii Kuchaiev. Tied-Lora: Enhacing parameter efficiency of LoRA with weight tying, November 2023.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving Thousands of Concurrent LoRA Adapters, November 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023.

Yiming Wang, Yu Lin, Xiaodong Zeng, and Guannan Zhang. MultiLoRA: Democratizing LoRA for Better Multi-Task Learning, November 2023.

Wenhan Xia, Chengwei Qin, and Elad Hazan. Chain of LoRA: Efficient Fine-tuning of Language Models via Residual Learning, January 2024.

A Related Works

Low-Rank Adaptation Hu et al. (2021) proposed Low-Rank Adaptation (LoRA) to fine-tune pre-trained models with low-rank adaptors. This method reduces the memory footprint by maintaining a low-rank weight adaptor for each layer. There are a few variants of LoRA proposed to enhance its performance (Renduchintala et al., 2023; Sheng et al., 2023; Xia et al., 2024), supporting multi-task learning (Wang et al., 2023), and further reducing the memory footprint (Dettmers et al., 2023). Lialin et al. (2023) proposed ReLoRA, a variant of LoRA designed for pre-training, but requires a full-rank training warmup to achieve comparable performance as the standard baseline.

Subspace Learning Recent studies have demonstrated that the learning primarily occurs within a significantly low-dimensional parameter subspace (Larsen et al., 2022; Gur-Ari et al., 2018). These findings promote a special type of learning called *subspace learning*, where the model weights are optimized within a low-rank subspace. This notion has been widely used in different domains of machine learning, including meta-learning and continual learning (Lee & Choi, 2018; Chaudhry et al., 2020).

Projected Gradient Descent GaLore is closely related to the traditional topic of projected gradient descent (PGD) (Chen & Wainwright, 2015; Chen et al., 2019). A key difference is that, GaLore considers the specific gradient form that naturally appears in training multi-layer neural networks (e.g., it is a matrix with specific structures). In contrast, traditional PGD mostly treats the objective as a general blackbox nonlinear function, and study the gradients in the vector space only.

Memory-Efficient Optimization There have been some works trying to reduce the memory cost of gradient statistics for adaptive optimization algorithms (Shazeer & Stern; Anil et al.; Dettmers et al., 2021). Adafactor (Shazeer & Stern) achieves sub-linear memory cost by factorizing the second-order statistics by a row-column outer product. GaLore shares similarities with Adafactor in terms of utilizing low-rank factorization to reduce memory cost, but GaLore focuses on the low-rank structure of the gradients, while Adafactor focuses on the low-rank structure of the second-order statistics. GaLore can reduce the memory cost for both first-order and second-order statistics, and can be combined with Adafactor to achieve further memory reduction. Quantization is also widely used to reduce the memory cost of optimizer states (Dettmers et al., 2021; Li et al., 2023). Furthermore, Lv et al. (2023) proposed fused gradient computation to reduce the memory cost of storing weight gradients during training.

In contrast to the previous memory-efficient optimization methods, GaLore operates independently as the optimizers directly receive the low-rank gradients without knowing their full-rank counterparts.

B DETAILS OF GALORE

Algorithm 1: Adam with GaLore

Input: A layer weight matrix $W \in \mathbb{R}^{m \times n}$ with $m \leq n$. Step size η, scale factor α, decay rates β_1, β_2, rank r, subspace change frequency T.
Initialize first-order moment $M_0 \in \mathbb{R}^{n \times r} \leftarrow 0$
Initialize second-order moment $V_0 \in \mathbb{R}^{n \times r} \leftarrow 0$
Initialize step $t \leftarrow 0$
repeat
  $G_t \in \mathbb{R}^{m \times n} \leftarrow -\nabla_W \varphi_t(W_t)$
  if $t \bmod T = 0$ **then**
    $U, S, V \leftarrow \text{SVD}(G_t)$
    $P_t \leftarrow U[:, :r]$ {Initialize left projector as $m \leq n$}
  else
    $P_t \leftarrow P_{t-1}$ {Reuse the previous projector}
  end if
  $R_t \leftarrow P_t^\top G_t$ {Project gradient into compact space}

  UPDATE(R_t) **by Adam**
    $M_t \leftarrow \beta_1 \cdot M_{t-1} + (1 - \beta_1) \cdot R_t$
    $V_t \leftarrow \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot R_t^2$
    $M_t \leftarrow M_t / (1 - \beta_1^t)$
    $V_t \leftarrow V_t / (1 - \beta_2^t)$
    $N_t \leftarrow M_t / (\sqrt{V_t} + \epsilon)$

  $\tilde{G}_t \leftarrow \alpha \cdot P N_t$ {Project back to original space}
  $W_t \leftarrow W_{t-1} + \eta \cdot \tilde{G}_t$
  $t \leftarrow t + 1$
until convergence criteria met
return W_t

We present the details of applying GaLore to Adam optimizer in Algorithm 1. The algorithm is similar to the original Adam optimizer, but with the gradient and update steps projected into a low-rank subspace. The low-rank subspace is updated every T steps, and the gradients are projected into the subspace determined by the left projector P_t and right projector Q_t. The update is then projected back to the original space using the same projectors.

C DETAILS OF PRE-TRAINING EXPERIMENT

C.1 ARCHITECTURE AND HYPERPARAMETERS

We introduce details of the LLaMA architecture and hyperparameters used for pre-training. Table 4 shows the most hyperparameters of LLaMA models across model sizes. We use a max sequence length of 256 for all models, with a batch size of 131K tokens. For all experiments, we adopt learning rate warmup for the first 10% of the training steps, and use cosine annealing for the learning rate schedule, decaying to 10% of the initial learning rate.

For all methods on each size of models (from 60M to 1B), we tune their favorite learning rate from a set of $\{0.01, 0.005, 0.001, 0.0005, 0.0001\}$, and the best learning rate is chosen based on the validation perplexity. We find GaLore is insensitive to hyperparameters and tends to be stable with the same learning rate across different model sizes. For all models, GaLore use the same hyperparameters, including the learning rate of 0.01, scale factor α of 0.25, and the subspace change frequency of T of 200. We note that since α can be viewed as a fractional learning rate, most of the modules (e.g., multi-head attention and feed-forward layers) in LLaMA models have the actual learning rate of 0.0025. This is, still, a relatively large stable learning rate compared to the full-rank baseline, which usually uses a learning rate ≤ 0.001 to avoid spikes in the training loss.

Table 4: Hyperparameters of LLaMA models for evaluation. Data amount are specified in tokens.

Params	Hidden	Intermediate	Heads	Layers	Steps	Data amount
60M	512	1376	8	8	10K	1.3 B
130M	768	2048	12	12	20K	2.6 B
350M	1024	2736	16	24	60K	7.8 B
1 B	2048	5461	24	32	100K	13.1 B
7 B	4096	11008	32	32	80K	10.5 B

C.2 Memory Estimates

As the GPU memory usage for a specific component is hard to measure directly, we estimate the memory usage of the weight parameters and optimizer states for each method on different model sizes. The estimation is based on the number of original parameters and the number of low-rank parameters, trained by BF16 format. For example, for a 60M model, LoRA ($r = 128$) requires 42.7M parameters on low-rank adaptors and $60M$ parameters on the original weights, resulting in a memory cost of 0.20G for weight parameters and 0.17G for optimizer states. Table 5 shows the memory estimates for weight parameters and optimizer states for different methods on different model sizes, as a compliment to the total memory reported in the main text.

Table 5: Memory estimates for weight parameters and optimizer states.

(a) Memory estimate of weight parameters.

	60M	**130M**	**350M**	**1B**
Full-Rank	0.12G	0.25G	0.68G	2.60G
GaLore	0.12G	0.25G	0.68G	2.60G
Low-Rank	0.08G	0.18G	0.36G	2.12G
LoRA	0.20G	0.44G	1.04G	4.74G
ReLoRA	0.20G	0.44G	1.04G	4.74G

(b) Memory estimate of optimizer states.

	60M	**130M**	**350M**	**1B**
Full-Rank	0.23G	0.51G	1.37G	5.20G
GaLore	0.13G	0.28G	0.54G	3.18G
Low-Rank	0.17G	0.37G	0.72G	4.25G
LoRA	0.17G	0.37G	0.72G	4.25G
ReLoRA	0.17G	0.37G	0.72G	4.25G

D Details of Fine-Tuning Experiment

We fine-tune the pre-trained RoBERTa-Base model on the GLUE benchmark using the model provided by the Hugging Face[1]. We trained the model for 30 epochs with a batch size of 16 for all tasks except for CoLA, which uses a batch size of 32. We use hyperparameters from Hu et al. (2021) for LoRA and tune the learning rate and scale factor for GaLore. Table 6 shows the hyperparameters used for fine-tuning RoBERTa-Base for GaLore.

E Additional Memory Measurements

We empirically measure the memory usage of different methods for pre-training LLaMA 1B model on C4 dataset with a token batch size of 256, as shown in Table 7.

[1] https://huggingface.co/transformers/model_doc/roberta.html

Table 6: Hyperparameters of fine-tuning RoBERTa base for GaLore.

	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B
Batch Size	16	16	16	32	16	16	16	16
# Epochs	30	30	30	30	30	30	30	30
Learning Rate	1E-05	1E-05	3E-05	3E-05	1E-05	1E-05	1E-05	1E-05
Rank Config.				$r = 4$				
GaLore α				4				
Max Seq. Len.				512				

	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B
Batch Size	16	16	16	32	16	16	16	16
# Epochs	30	30	30	30	30	30	30	30
Learning Rate	1E-05	2E-05	2E-05	1E-05	1E-05	2E-05	2E-05	3E-05
Rank Config.				$r = 8$				
GaLore α				2				
Max Seq. Len.				512				

Table 7: Measuring memory and throughput on LLaMA 1B model.

Model Size	Layer Wise	Methods	Token Batch Size	Memory Cost	Throughput	
					#Tokens / s	#Samples / s
1B	✘	AdamW	256	13.60	1256.98	6.33
		Adafactor	256	13.15	581.02	2.92
		Adam8bit	256	9.54	1569.89	7.90
		8-bit GaLore	256	7.95	1109.38	5.59
1B	✔	AdamW	256	9.63	1354.37	6.81
		Adafactor	256	10.32	613.90	3.09
		Adam8bit	256	6.93	1205.31	6.07
		8-bit GaLore	256	5.63	1019.63	5.13