

GaLore: Gradient Low-Rank Projection

Stanford CS224N {Default} Project

Chungsuen Stephen Chan
Department of Computer Science
Stanford University
schan99@stanford.edu

Abstract

Training Large Language Models (LLMs) presents significant challenges due to the growing parameter count and optimizer states. The parameter count growth rate far exceeds the growth rate of the GPU hardware memory capacity. Consequently, modern large language model training requires sophisticated infrastructure to distribute the workload across multiple GPU. The amount of computing resources and the sophistication required puts it out of reach for most organizations.

Researchers have tried to reduce the GPU memory footprint with various parameter-efficient fine-tuning (PEFT) methods. One popular memory-reduction approach, low-rank adaptation (LoRA [1]), adds a trainable low-rank matrix to the frozen pre-trained weight in each layer. However, this approach may underperform training with full-rank weights since it limits the parameter search to a low-rank subspace and alters the training dynamics [2], and further, may require a full-rank warm start [3].

This experiment looks into a PEFT approach called GaLore [[4]] which claimed to significantly reduce the GPU memory footprint of the optimizer states while maintaining the efficiency and quality for large-scale LLM pre-training and post-training. According to the paper, they are the first to pre-train the LLaMA 7B model from scratch on a consumer-grade GPU with only 22GB memory by using a combination of GaLore and other existing memory-reduction techniques.

This experiment implemented the GaLore algorithm from scratch and benchmarked the implementation vs. AdamW by running LLM fine-tuning tasks. The results showed that GaLore, without any additional memory saving technique, reduced the peak GPU memory requirement by up to 30%. The results also showed that the GaLore hyperparameter T had a "sweet spot" where the execution time was within 10% while model quality was within 5% of AdamW. Depending on the ML task, sometimes GaLore executed faster than AdamW while delivering higher quality trained models.

1 Key Information to include

- Mentor: Caroline Choi
- External Collaborators (if you have any): No
- Sharing project: No
- Late Days (if you are using any): 3

Team member	Late days remaining
Chungsuen Stephen Chan	3
Total pooled	3
Team size	1
Late days applied for the report	3

2 Introduction

Training Large Language Models (LLMs) presents significant challenges because the model parameters and the optimizer states demand a lot of GPU memory. The number of model parameters has grown from 1.5 billion in GPT2 to 1.8 trillion in GPT4 while GPU memory only grew from 32GB (Volta) to 288GB (Rubin). Researchers have to use multiple GPU and invent methods to distribute and coordinate the workload. Traditional approaches designed for big data or high-performance computing cannot fully utilize the GPU resources and new methods have to be developed to meet the new requirements [5]. While a few well-funded organizations have the resources to handle it, most cannot afford the resources. This memory footprint bottleneck prevents wider adaptation of LLM and makes it difficult for many people to experience and contribute to the technology.

This problem will only get worse. Empirical scaling law [6] shows that there are power-law relationships where AI model performance improves as compute, dataset size, and parameter counts increase. Scaling laws predict that, up to a certain threshold, cross-entropy loss decreases as model parameters, dataset size, and compute increase [7]. Consequently, we can expect model parameter count to keep increasing in the foreseeable future. Meeting the increasing demand of GPU memory is an uphill battle.

Researchers have invented various parameter-efficient fine-tuning (PEFT) methods [8] to address the issue (the appendix includes a taxonomy of the PEFT approaches). Among these approaches, LoRA [1] is one of the most popular approach. LoRA reparametrizes 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 low-rank adapters that are updated during training. Since $r \ll \min(m, n)$, A and B have far fewer trainable parameters and thus smaller GPU memory footprint. While LoRA can deliver results close to full-rank training in RL and fine-tuning [9], in other situations LoRA may underperform since it limits the model parameter search to a low-rank subspace and alter the training dynamics [2], and further, may require a full-rank warm start[3].

The optimizer is an indispensable part of training and fine-tuning. First-order optimization methods like AdamW [10] has many advantages [11] over Stochastic Gradient Descent optimizers [12] because it decouples weight decay from the gradient update, resulting in better regularization, improved generalization, and more stable training for fast convergence. The trade-off is that AdamW stores the first and second momentum of model parameters in GPU memory (see Appendix "Estimated memory consumption of pre-training"). Consequently, as the number of model parameters grows, the GPU memory is consumed even more quickly. Reducing the optimizer states memory footprint is an important challenge.

A recent PEFT approach, GaLore [4], reduces the optimizer memory footprint by converting the parameter gradient to a low-rank projection so it requires less memory. This approach works because parameter gradients are "naturally" low-rank [13] and can be converted into a low-rank projection, and it applies to both pre-training and post-training. The key advantage of this approach is that it has no restriction on the model parameter optimization search space, thus it supports full-parameter learning but is more memory-efficient. The paper showed a list of tasks where GaLore achieved better performance than LoRA with less memory footprint (see appendix "Evaluating GaLore on GLUE benchmark").

GaLore can be tuned with a few hyperparameters to adapt to different workloads. GaLore generates the low-rank projection using SVD calculation on the full-rank gradient. To manage the SVD calculation execution time, GaLore introduced a hyperparameter, subspace change frequency (T), so the SVD is recalculated from the current full-rank gradient only once every T steps. If not recalculated, GaLore reuses the same low-rank projection from the previous SVD calculation. According to [4] this is a trade-off between run-time performance vs. perplexity. Since LoRA does not perform SVD calculation, this execution-time overhead is an important disadvantage of GaLore. But if it can be tuned to limit the damage, then GaLore can still be an attractive approach. The paper suggested that a sweet spot exists for the value of the hyperparameter T : When T reaches 100, the perplexity does not deteriorate any more until $T = 10000$. If this works then T can be simply set to a 10000 to minimize the SVD calculation overhead. In the following graph, T is "Update Frequency".

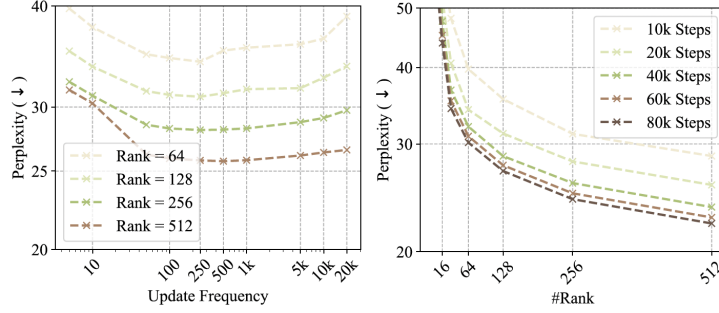


Figure 1: Ablation study of GaLore on 130M models. Left: varying subspace subspace change frequency T . Right: varying subspace rank and training iterations. From [4]

The hyperparameter rank r is a constant applied in the generation of the low-rank projectors of the gradient. If the full-rank gradient is $G \in \mathbb{R}^{m \times n}$, the left projector is $P \in \mathbb{R}^{m \times r}$ and the right projector is $Q \in \mathbb{R}^{n \times r}$. The consideration for r is similar to that in LoRA.

The hyperparameter scale factor α controls the strength of the low-rank update, which is similar to the scale factor α/r appended to the low-rank adapter in LoRA. The paper described that the algorithm is very robust to changes in α , with values between 0.2 and 2.0 found to work well.

This experiment evaluated GaLore using the default project as the baseline. I implemented the GaLore algorithm and confirmed that it only required changes to the gradient projection logic inside the AdamW step function. The results showed that, without using other memory-saving techniques, GaLore reduced GPU memory allocation by up to 30%. The results also showed that the "sweet spot" of the hyperparameter T existed, which helped to reduce the execution-time overhead without sacrificing much of the quality of model, although there were some task-specific exceptions.

3 Related Work

3.1 Low-Rank Adaptation

LoRA [1] proposes using low-rank adapters to fine-tune language models. By maintaining a low-rank weight adapter for each layer, LoRA reduces the amount of memory required. The trade-off is that the model parameter optimization is restricted to a low rank subspace so the end result may not match that of a full-rank training. Nevertheless, LoRA is a significant improvement and it achieved rapid adoption. Other researchers followed up and propose various enhancements on top of LoRA. These variants include Chain of LoRA (COLA) [2], QLoRA [14], DoRA [15], LoRA+ [16], AdaLORA [17]. However, each approach has its own trade-off. For example, ReLORA [3] requires a full-rank training warm-up to match the results of the standard baseline.

3.2 Optimizer State Minimization

The fused backward operation introduced by LOMO [18] reduces memory usage during training. The standard LOMO optimizer is fundamentally a modified SGD. It reduces memory by fusing gradient computation and parameter updates, immediately discarding gradients after use. When combined with the standard SGD optimizer, LOMO eliminates the memory cost typically required for storing both optimizer states and gradients. Because LOMO stores zero optimizer states (no momentum or variance), it cannot perform the adaptive updates required by AdamW. To provide the benefits of AdamW while maintaining low memory, researchers introduced AdaLOMO [19] which uses Non-negative Matrix Factorization (NMF) to store the second-order moment estimation in a highly compressed way. It adds an adaptive learning rate for each parameter while still using the fused backward operation. This allows AdaLOMO to maintain very low optimizer memory usage while benefiting from adaptive optimization. Although LOMO and AdaLOMO are important advances for memory-efficient optimization especially for fine-tuning or continual pre-training, they may not work as well for training models from scratch at very large scales. For instance, AdaLOMO

relies on the vanilla Adafactor optimizer, which has been shown to cause greater training instability when used at larger scales.

3.3 Memory-Efficient Optimization

Outside of low-rank, there have been various approaches to reduce the memory requirements of model training and fine-tuning. Quantization [20] is a technique that reduces the precision of a model’s numerical values (e.g., converting 32-bit floating-point weights to 8-bit integers) to decrease model size, speed up inference, and lower memory usage. It maps high-precision, continuous values to a smaller set of discrete, lower-precision values. Layer-wise weight update which combines weight update during back-propagation [19] [18] reduces the optimizer memory footprint by only keeping the optimizer data of a layer in memory instead of all layers. FRUGAL [21] reduces memory usage by splitting gradient updates into low-dimensional updates with advanced optimizers (e.g., AdamW) and using state-free methods (e.g., SignSGD and SGD) for the rest. These techniques can be combined with GaLore to achieve more GPU memory saving.

3.4 Low-Rank Gradient

Deep neural networks inherently learn low-rank structures in their weights during machine training [22]. Gradient updates of model parameter weights in deep learning, particularly within large language models (LLMs), reside within a low-dimensional subspace that captures the majority of gradient energy, and this subspace tends to evolve slowly over training iterations [23]. This is the observation that allows various low-rank ideas to be applied to machine learning. WeLore [24] categorizing weight matrices of pretrained models into two broad categories - LRCs and N-LRCs based on their low-rank structure. It unifies weight compression and memory-efficient fine-tuning into one, in a data-agnostic and one-shot manner.

4 Approach

Here is the standard AdamW algorithm (from the default project handout):

Algorithm 1 Adam algorithm. g_t^2 indicates the element-wise square $g_t \odot g_t$. All operations on vectors are element-wise. With B_1^t and B_2^t , we denote B_1 and B_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize time step)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective function at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$

return θ_t (Resulting parameters)

Note, at the expense of clarity, there is a more *efficient version* of the above algorithm where the last three lines in the loop are replaced with the following two lines: $\alpha_t \leftarrow \alpha \cdot \sqrt{1 - \beta_2^t} / (1 - \beta_1^t)$ and $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t / (\sqrt{v_t} + \epsilon)$

Figure 2: Standard AdamW algorithm (from the default project)

GaLore replaces the steps inside the red dotted rectangle above with the following steps:

$U\Sigma V^T \leftarrow SVD(g_t)$	(Use SVD to compute the U and V vectors every T calls to the step function, otherwise reuse the same U and V from before)
$L_t \leftarrow \begin{cases} g_t \cdot V^T & \text{if } g_t \text{ is tall } (m > n) \\ U^T \cdot g_t & \text{if } g_t \text{ is wide } (m < n) \end{cases}$	(computes a low-rank gradient from $g_t \in \mathbb{R}^{m \times n}$)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot L_t$	(same as before but using the low-rank gradient)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot L_t^2$	(same as before but using the low-rank gradient)
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$	(same as before but using the low-rank gradient)
$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$	(same as before but using the low-rank gradient)
$n_t \leftarrow \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$	(the last step $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ is split into three steps)
$H_t \leftarrow \begin{cases} n_t \cdot V^T \cdot s & \text{if } g_t \text{ is tall} \\ U^T \cdot n_t \cdot s & \text{if } g_t \text{ is wide} \end{cases}$	(convert back into high-rank)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot H_t$	(complete the last step)

Figure 3: GaLore algorithm changes

In the GaLore algorithm above, step 1 ($U\Sigma V^T \leftarrow SVD(g_t)$) is time-consuming. GaLore supports a hyperparameter T such that this step is executed only once every T -steps. The logic is like this (where t is time-step in the standard AdamW algorithm and $\%$ is the modulo operator):

if ($U\Sigma V^T = \text{None}$) or ($t \% T = 0$):
 $U\Sigma V^T \leftarrow SVD(g_t)$

The hyperparameter r is used in the $SVD(g_t)$ calculation above to determine the dimensions of U and V . The hyperparameter α is used in the last step $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot H_t$. See Appendix "GaLore Hyperparameters" for more details of how the hyperparameters T , r , and α are used.

The GaLore paper benchmarked its performance after combining with other techniques such as 8-bit optimizer and per-layer weight updates for further memory-efficiency. In order to measure the contribution of the GaLore algorithm alone, no other memory-reduction techniques were included in this experiment.

The original paper has an implementation in the Python package `galore_torch` which was an AdamW optimizer with embedded GaLore logic. Since I wanted to ensure that all differences were due to the GaLore algorithm and not their AdamW implementation (there are variants of AdamW), I did not use their package for this experiment. Instead, I chose the "clean room" approach where I started with the default project and added the GaLore logic inside the AdamW step function using `torch.linalg.svd` for the SVD calculation (the new code is referred to as "MyGaLore" in the following sections). To ensure the correctness of MyGaLore logic, I compared the computed gradients of MyGaLore with those of `galore_torch` during unit-testing and that was the only time their package was used.

Then I made a copy of each ML task (Sonnet Generation, Paraphrase Detection, Sentiment Classification) to call MyGaLore. In the following sections, these new copies are referred to as the "MyGaLore ML task" vs. the "baseline ML task" of the default project before the changes.

For GPU memory saving, the focus was on the "peak" GPU memory allocation which determines if a ML task could run on a GPU. I executed each baseline ML task and the corresponding MyGaLore ML task once with $T=10000$ (GPU memory allocation did not depend on T so any value will work) and then compared their peak GPU memory allocation to calculate the saving. These steps were repeated for every ML task and every language model to observe the effects of the language model size.

To confirm the "sweet spot" of the hyperparameter T , I executed each MyGaLore ML task multiple times using various values of T from 50 to 100000 (which covered the "sweet spot" range mentioned in the paper) and measured how the execution time and the quality of the generated models changed with T . I also compared these metrics with those of the corresponding baseline ML task to measure the differences. These steps were repeated for every ML task and every language model to observe the effects of the language model size.

5 Experiments

5.1 Data

This experiment executed the default project ML tasks (Sentiment Classifier, Sonnet Generation, and Paraphrase Detection) which used the default project data sets. The data set sizes are:

	Sentiment Classification		Paraphrase Detection	Sonnet Generation
<i>Dataset Name</i>	<i>SST</i>	<i>cfimdb</i>	<i>Quora</i>	
train	8,544	1,701	141,506	143
dev	1,101	245	20,215	
test	2,210	488	40,431	12

5.2 Evaluation method

ML tasks were measured on these metrics:

Peak GPU memory allocation This is the maximum amount of GPU memory allocated. The ML task called `torch.cuda.reset_peak_memory_stats` to reset the counter before the training loop and the testing loop. After each loop, it called `torch.cuda.max_memory_allocated` to print the peak GPU memory allocated in the loop.

Execution time This is the wall-clock elapsed time for the full execution of the ML task.

Quality of model The quality of the model generated from the training is either the CHrF score (for Sonnet Generation) or the accuracy (for Paraphrase Detection and Sentiment Classification).

Metric	Peak GPU memory	execution time	quality of model
Sentiment Classification	average of SST train, SST test, cfimdb train, cfimdb test peak GPU	total time of SST train, SST test, cfimdb train, cfimdb test	average of SST dev acc and cfimdb dev acc
Paraphrase Detection	average of train and test peak GPU	total time of train and test	dev paraphrase acc
Sonnet Generation	average of train and test peak GPU	total time of train and test	generated sonnets dev CHrF / 100.0

5.3 Experimental details

MyGaLore was applied to all `nn.Linear` parameter layers using the same r and α . Tests were repeated on three language models: `gpt2`, `gpt2-medium`, `gpt2-large`. The learning rate was $1e-5$. For the Sentiment Classifier, `fine_tune_mode='full-model'`. The GaLore hyperparameters were:

GaLore Hyperparameter	Value(s)
T	50, 100, 1000, 5000, 10000, 100000
r	16
α	2.0

All other parameters were left at their default values from the default project. All tests were performed on a GCP G2 instance with an NVIDIA L4 GPU unless otherwise noted. I executed the following steps and recorded the peak GPU memory allocation, execution time, and quality of model of each execution for analysis. Note that Sentiment Classification only loads `gpt2`.

For each ML task in Sentiment Classification, Paraphrase Detection, Sonnet Generation:

For each language model on `gpt2`, `gpt2-medium`, `gpt2-large`:

Execute baseline ML task

For each T in 50, 100, 1000, 5000, 10000, 100000:

Execute MyGaLore ML task with T

5.4 Results

5.4.1 Leaderboard Scores

Task	Leaderboard Score
Paraphrase Detection	0.859
Sonnet Generation	41.888

5.4.2 GPU Memory Saving

MyGaLore tasks required less peak GPU memory allocation than the corresponding baseline ML task for every language model and every ML Task. Larger language models have higher saving percentages. The MyGaLore measurement was from $T = 10000$ (the results are the same regardless of T). GPU memory saving percentage = $1 - \text{MyGaLore peak GPU memory} / \text{baseline peak GPU memory}$:

ML Task	gpt2	gpt2-medium	gpt2-large
Sentiment Classifier	15%		
Paraphrase Detection	6%	20%	25% ¹
Sonnet Generation	17%	25%	30%

5.4.3 Hyperparameter T vs. Execution Time

MyGaLore tasks were executed with the following values for the hyperparameter T : 50, 100, 1000, 5000, 10000, 100000. In the following graphs, the horizontal axis are T , the vertical axis are execution time. Each language model has a pair of same color lines: solid lines are MyGaLore, dotted lines are baseline.

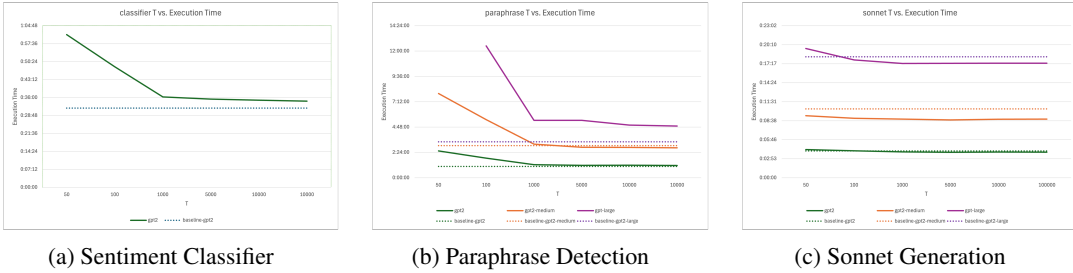


Figure 4: T vs. Execution Time

MyGaLore tasks executed slower than baseline tasks when the value of T was low (i.e., high subspace change frequency). As T increased, SVD calculations became less frequent and MyGaLore execution time decreased. When $T \geq 100$, MyGaLore Sentiment Classifier (on gpt2) was 10% slower than the baseline and stayed flat as T increased. MyGaLore Paraphrase Detection execution time improved as T increased above 100: At $T = 10000$ it was 10% slower on gpt2, 6% faster on gpt2-medium, and 47% slower on gpt2-large than baseline. MyGaLore Sonnet Generation execution time was 15% faster (on gpt2-medium) and 5% faster (on other models) than baseline. These execution times were stable even when T increased to 100000. All tasks, both baseline and MyGaLore, executed slower on larger language models.

5.4.4 Hyperparameter T vs. Quality

The following graphs compare how the quality of the generated models changed when T changed.

¹The baseline Paraphrase Detection with gpt2-large failed to load into a L4 GPU. I ran it again using an A100 GPU to obtain its peak GPU memory allocation. Using MyGaLore, I was able to run Paraphrase Detection with gpt2-large on an L4 GPU with various values of T .



Figure 5: T vs. Quality

Generally speaking, the MyGaLore quality of model dropped when T increased from 50 to 100 and then improved and stabilized as T continued to increase until 10000. For Sentiment Classifier (on gpt2), the quality ranged from 1% to 3% above the baseline depending on T . For Paraphrase Detection the quality was consistently 4% below the baseline on all language models. For Sonnet Generation, the quality was consistently between 4% below (on gpt2) to 1% above (on gpt2-large) the baseline. When T increased from 10000 to 100000, the quality started to deteriorate slightly for all ML tasks. This range was consistent with the "sweet spot" of T indicated in the GaLore paper. Sometimes MyGaLore quality was better than that of baseline: At $T = 10000$, Sentiment Classifier on gpt2 was 2% above the baseline and Sonnet Generation on gpt2-large was 1% above the baseline.

6 Analysis

GaLore can be cleanly implemented on top of standard AdamW. The experiment showed that GaLore reduced peak GPU memory allocation by up to 30%. Larger language models had more linear layers and had higher savings percentage. Since GPU memory is also consumed by activations and temporary variables, the actual savings percentage depends not only on the model parameter count but also on other factors [25]. The dimensions of the low-rank projection generated by GaLore was controlled by r so GPU memory savings are independent of T .

Regarding the effects of the hyperparameter T , as T increased to 100, GaLore execution time and quality metric stabilized close to the baseline until $T = 10000$. As T increased from 10000 to 100000, the quality of model started to deteriorate although the execution times were still stable. These results confirmed the "sweet spot" in the range of $100 \leq T \leq 10000$ where the execution time and the quality of model stabilized close to the baseline, which makes T easy to tune. However, there were some task-specific exceptions. For example, Paraphrase Detection on gpt2-large executed much slower than the baseline although the quality of model was within 5% of the baseline.

Occasionally, GaLore executed faster than the baseline and generated models with better quality than the baseline. One possible explanation is that GaLore performed fewer FLOP per optimizer step than AdamW so it executed faster. The GaLore low-rank projection naturally enforced a simpler structure on the gradient updates and acted as a form of regularization that avoided overfitting and improved generalization on unseen data so the quality was better. However, this observation was not universal so there could also be some task-specific and language model-specific factors involved.

7 Conclusion

Handling GPU memory demand is challenging but necessary for the continued development of LLM. GaLore is a promising PEFT approach that can be readily implemented on leading optimizers like AdamW. For some ML tasks, GaLore can be tuned to deliver GPU memory savings without sacrificing a lot of execution-time and quality of the generated model.

More research is needed to determine how other GaLore hyperparameters should be tuned, which layers should GaLore be applied to, and if GaLore shifts the optimal learning rate from the baseline. Since this experiment only included fine-tuning tasks, it would be interesting to test other tasks mentioned in the GaLore paper (see appendix "Evaluating GaLore on GLUE benchmark"). GaLore may also be applicable to non-LLM tasks such as image deep learning tasks.

Team contributions (Required for multi-person team)

Not Applicable

References

- [1] Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *ICLR*, 2022.
- [2] Wenhan Xia, Chengwei Qin, and Elad Hazan. Chain of lora: Efficient fine-tuning of language models via residual learning. In *ICML*, 2024.
- [3] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *ICLR*, 2024.
- [4] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. In *ICML*, 2024.
- [5] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision, 2022.
- [6] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.
- [7] Yangyi Chen, Binxuan Huang, Yifan Gao, Zhengyang Wang, Jingfeng Yang, and Heng Ji. Scaling laws for predicting downstream performance in llms, 2025.
- [8] Jieming Bian, Yuanzhe Peng, Lei Wang, Yin Huang, and Jie Xu. A survey on parameter-efficient fine-tuning for foundation models in federated learning, 2025.
- [9] John Schulman and Thinking Machines Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. <https://thinkingmachines.ai/blog/lora/>.
- [10] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.
- [11] Ruslan Abdulkadirov, Pavel Lyakhov, and Nikolay Nagornov. Survey of optimization algorithms in modern neural networks. *Mathematics*, 11(11), 2023.
- [12] Bin Shi. On the hyperparameters in stochastic gradient descent with momentum, 2022.
- [13] Yuandong Tian, Lantao Yu, Xinlei Chen, and Surya Ganguli. Understanding self-supervised learning with dual deep networks, 2021.
- [14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.
- [15] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation, 2024.
- [16] Soufiane Hayou, Nikhil Ghosh, and Bin Yu. Lora+: Efficient low rank adaptation of large models, 2024.
- [17] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning, 2023.
- [18] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources, 2024.
- [19] Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. Adalomo: Low-memory optimization with adaptive learning rate, 2024.

- [20] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization, 2022.
- [21] Philip Zmushko, Aleksandr Beznosikov, Martin Takáč, and Samuel Horváth. Frugal: Memory-efficient optimization by reducing state overhead for scalable training, 2025.
- [22] Laura Balzano, Tianjiao Ding, Benjamin D. Haeffele, Soo Min Kwon, Qing Qu, Peng Wang, Zhangyang Wang, and Can Yaras. An overview of low-rank structures in the training and adaptation of large models, 2026.
- [23] Can Yaras, Peng Wang, Laura Balzano, and Qing Qu. Compressible dynamics in deep overparameterized low-rank learning and adaptation, 2026.
- [24] Ajay Jaiswal, Yifan Wang, Lu Yin, Shiwei Liu, Runjin Chen, Jiawei Zhao, Ananth Grama, Yuandong Tian, and Zhangyang Wang. From low rank gradient subspace stabilization to low-rank weights: Observations, theories, and applications, 2025.
- [25] Xiaoxuan Liu, Siddharth Jha, and Alvin Cheung. An evaluation of memory optimization methods for training neural networks, 2023.
- [26] Shangshang Wang, Julian Asilis, Ömer Faruk Akgül, Enes Burak Bilgin, Ollie Liu, and Willie Neiswanger. Tina: Tiny reasoning models via lora, 2025.

A Appendix

A.1 PEFT Categories

PEFT is an active field of research. Here is a taxonomy of the various approaches [8]:

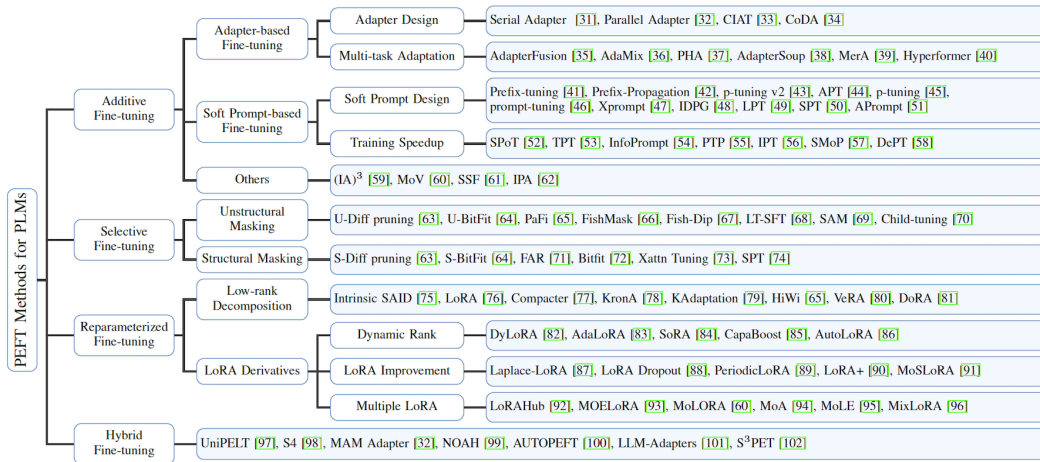


Figure 6: Taxonomy of Parameter-Efficient Fine-Tuning Methods for Large Models reprinted. From [8]

These approaches can be broadly categorized into Additive, Selective, Re-parameterized and Hybrid.

- Additive methods work by freezing the original backbone model parameters and adding new, trainable layers or parameters. They are highly modular (layers can be swapped), effective for low-resource training, and preserves original model knowledge. However, they introduce slight inference latency due to the additional layers.
- Selective methods work by freezing the majority of the model parameters and tunes only a small, specific subset of them, such as bias terms or top layers. There is no additional parameters, making it very efficient during inference. But they can be computationally intensive during training due to sparse updates.

- Reparametrized methods use low-rank decomposition to represent updates, training a small number of parameters that are then integrated back into the original weights. There is no inference latency (weights are merged). The parameter efficiency is high, and performance often equals full fine-tuning.
- Hybrid methods combine multiple PEFT methods to leverage their respective strengths. They often provide superior performance and flexibility by mixing techniques to handle different tasks at the expense of higher complexity in design and implementation.

A.2 GaLore Hyperparameters

A.2.1 Hyperparameter Subspace Change Frequency T

In the GaLore algorithm, step 1 $U\Sigma V^T \leftarrow SVD(g_t)$ is a time-consuming step. Therefore GaLore provides a hyperparameter T such that this step is executed only once every T -steps. The logic is like this (t is time-step in the original AdamW algorithm):

```
if (U\Sigma V^T is None) or (t % T == 0):
    U\Sigma V^T ← SVD(g_t)
```

The GaLore paper measured the relationship between T and perplexity and reported a "sweet spot" for $T \geq 100$ which simplified the trade-off. According to [4] there is a trade-off between run-time performance vs. perplexity:

"A very frequent subspace change increases the overhead (since new P_t and Q_t need to be computed) and breaks the condition of constant projection. In practice, it may also impact the fidelity of the optimizer states, which accumulate over multiple training steps. On the other hand, a less frequent change may make the algorithm stuck into a region that is no longer important to optimize."

A.2.2 Hyperparameter Rank r

The hyperparameter rank r is a fixed value for every parameter layer initialized when the optimizer is instantiated. In the GaLore algorithm above, step 2 generates the low-rank gradient L_t from the full-rank gradient g_t where $g_t \in \mathbb{R}^{m \times n}$:

$$L_t = \begin{cases} g_t \dot{V}^T & \text{if } g_t \text{ is tall } (m \geq n) \\ U^T \dot{g}_t & \text{if } g_t \text{ is wide } (m < n) \end{cases}$$

The hyperparameter rank is used in the calculation above like this:

```
u, s, v = torch.linalg.svd(full_rank_gradient, full_matrices=False)
if full_rank_gradient is tall: # (m > n)
    low_rank_gradient = full_rank_gradient @ v[:rank, :].t()
else:
    low_rank_gradient = u[:, :rank].t() @ full_rank_gradient
```

GaLore low-rank projection can help to filter out "noise" in the data to produce better quality model. A small rank can also reduce the number of FLOP, thus reducing the execution time. Some research showed that rank 16 delivers the best performance for some tasks [26].

A.2.3 Hyperparameter scale factor α

The hyperparameter scale factor α is a fixed value for every parameter layer initialized when the optimizer is instantiated. It is used when the low-rank projection is converted back to high-rank:

$$H_t = \begin{cases} n_t \dot{V}^T \dot{scale} & \text{if } g_t \text{ is tall } (m \geq n) \\ U^T \dot{n}_t \dot{scale} & \text{if } g_t \text{ is wide } (m < n) \end{cases}$$

Here it is designated as *scale* instead of α to avoid confusion with the step-size. According to the GaLore paper, α does not depend on the rank r because when r is small during pre-training, α/r significantly affects the convergence rate, unlike fine-tuning.

A.3 Evaluating GaLore on GLUE benchmark

The GaLore paper [4] showed a list of tasks where GaLore achieved better performance than LoRA with less memory footprint.

	Memory	CoLA	STS-B	MRPC	RTE	SST2	MNLI	QNLI	QQP	Avg
Full Fine-Tuning	747M	62.24	90.92	91.30	79.42	94.57	87.18	92.33	92.28	86.28
GaLore (rank=4)	253M	60.35	90.73	92.25	79.42	94.04	87.00	92.24	91.06	85.89
LoRA (rank=4)	257M	61.38	90.57	91.07	78.70	92.89	86.82	92.18	91.29	85.61
GaLore (rank=8)	257M	60.06	90.82	92.01	79.78	94.38	87.17	92.20	91.11	85.94
LoRA (rank=8)	264M	61.83	90.80	91.90	79.06	93.46	86.94	92.25	91.22	85.93

Figure 7: Evaluating GaLore for memory-efficient fine-tuning on GLUE benchmark using pre-trained RoBERTa-Base. We report the average score of all tasks. From [4]

A.4 Estimated memory consumption of pre-training

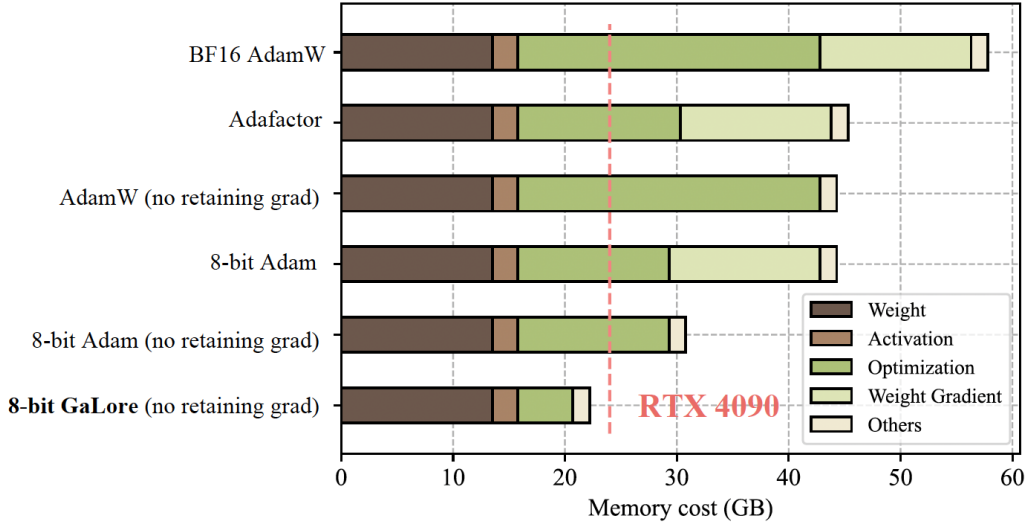


Figure 8: Estimated memory consumption of pre-training a LLaMA 7B model with a token batch size of 256 on a single device, without activation checkpointing and memory offloading. From [4]