# GaLore: Memory-Efficient LLM Training by Natural Gradient Low-Rank Projection

**Jiawei Zhao** [1] **Zhenyu Zhang** [3] **Beidi Chen** [2 4] **Zhangyang Wang** [3] **Anima Anandkumar** [* 1] **Yuandong Tian** [* 2]

## 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. 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. Code is provided in the link.

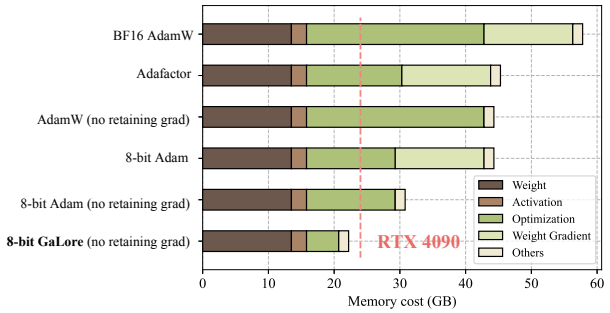


Figure 1: Estimated memory consumption of pre-training a LLaMA 7B model with a token batch size of 256 on a single device, without activation checkpointing and memory offloading[2]. Details refer to Section 5.5.

## 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 (**???**). 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 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 (**?**), memory offloading (**?**), 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.

[*]Equal advising [1]California Institute of Technology [2]Meta AI [3]University of Texas at Austin [4]Carnegie Mellon University. Correspondence to: Jiawei Zhao <jiawei@caltech.edu>, Yuandong Tian <yuandong@meta.com>.

*Proceedings of the $41^{st}$ International Conference on Machine Learning*, Vienna, Austria. PMLR 235, 2024. 

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

[2]In the figure, "no retaining grad" denotes the application of per-layer weight update to reduce memory consumption of storing weight gradient (**?**).

**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
```

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 (**?**). Among them, the popular Low-Rank Adaptation (LoRA **?**) *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 (**?**).

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 (**?**). For pre-training from scratch, it is shown to require a full-rank model training as a warmup (**?**), 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.

**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.

We first show theoretically 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, 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.

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. Our experiment (Fig. 3) shows that it works for popular optimizers such as AdamW, 8-bit Adam, and Adafactor. In addition, its performance is insensitive to very few hyper-parameters it introduces. We also provide theoretical justification on the low-rankness of gradient update, as well as the convergence analysis of GaLore.

## 2. Related Works

**Low-rank adaptation.** **?** 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 (**????**), supporting multi-task learning (**?**), and further reducing the memory footprint (**?**). **?** 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. Inspired by LoRA, **?** also suggested that gradients can be compressed in a low-rank subspace, and they proposed to use random projections to compress the gradients. There have also been approaches that propose training networks with low-rank factorized weights from scratch (**???**).

**Subspace learning.** Recent studies have demonstrated that the learning primarily occurs within a significantly low-dimensional parameter subspace (**??**). 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 (**??**).

**Projected gradient descent.** GaLore is closely related to the traditional topic of projected gradient descent (PGD) (**??**). 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), proving many of its properties (e.g., Lemma 3.3, Theorem 3.2, and Theorem 3.8). In contrast, traditional PGD mostly treats the objective as a general blackbox nonlinear function, and study the gradients in the vector space only.

**Low-rank gradient.** Gradient is naturally low-rank during training of neural networks, and this property have been studied in both theory and practice (**???**). It has been applied to reduce communication cost (**??**), and memory footprint during training (**???**).

**Memory-efficient optimization.** There have been some works trying to reduce the memory cost of gradient statistics for adaptive optimization algorithms (**???**). Quantization is widely used to reduce the memory cost of optimizer states (**??**). Recent works have also proposed to reduce weight gradient memory by fusing the backward operation with the optimizer update (**??**).

## 3. GaLore: Gradient Low-Rank Projection

### 3.1. Background

**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 ($\eta$ is the learning rate):

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

where $\tilde{G}_t$ is the final processed gradient to be added to the weight matrix and $\rho_t$ is an entry-wise stateful gradient regularizer (e.g., Adam). The state of $\rho_t$ 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 $\tilde{G}_t$:

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

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

$$\tilde{G}_t = M_t / \sqrt{V_t + \epsilon} \quad (4)$$

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

**Low-rank updates.** For a linear layer $W \in \mathbb{R}^{m\times n}$, LoRA and its variants utilize the low-rank structure of the update matrix by introducing a low-rank adaptor $AB$:

$$W_T = W_0 + B_T A_T, \quad (5)$$

where $B \in \mathbb{R}^{m\times r}$ and $A \in \mathbb{R}^{r\times n}$, and $r \ll \min(m,n)$. $A$ and $B$ are the learnable low-rank adaptors and $W_0$ is a fixed weight matrix (e.g., pre-trained weight).

### 3.2. Low-Rank Property of Weight Gradient

While low-rank updates are proposed to reduce memory usage, it remains an open question whether the weight matrix should be parameterized as low-rank. In many situations, this may not be true. For example, in linear regression $\boldsymbol{y} = W\boldsymbol{x}$, if the optimal $W^*$ is high-rank, then imposing a low-rank assumption on $W$ never leads to the optimal solution, regardless of what optimizers are used.

Surprisingly, while the weight matrices are not necessarily low-rank, the gradient indeed becomes low-rank during the training for certain gradient forms and associated network architectures.

**Reversible networks.** Obviously, for a general loss function, its gradient can be arbitrary and is not necessarily low rank. Here we study the gradient structure for a general family of nonlinear networks known as "reversible networks" (**?**), which includes not only simple linear networks but also deep ReLU/polynomial networks:

**Definition 3.1** (Reversiblity (**?**))**.** A network $\mathcal{N}$ that maps input $\boldsymbol{x}$ to output $\boldsymbol{y} = \mathcal{N}(\boldsymbol{x})$ is *reversible*, if there exists $L(\boldsymbol{x}; W)$ so that $\boldsymbol{y} = L(\boldsymbol{x}; W)\boldsymbol{x}$, and the backpropagated gradient $\boldsymbol{g_x}$ satisfies $\boldsymbol{g_x} = L^\top(\boldsymbol{x}; W)\boldsymbol{g_y}$, where $\boldsymbol{g_y}$ is the backpropagated gradient at the output $\boldsymbol{y}$. Here $L(\boldsymbol{x}; W)$ depends on the input $\boldsymbol{x}$ and weight $W$ in the network $\mathcal{N}$.

Please check Appendix B.1 for its properties. For reversible networks, the gradient takes a specific form.

**Theorem 3.2** (Gradient Form of reversible models)**.** *Consider a chained reversible neural network $\mathcal{N}(\boldsymbol{x}) := \mathcal{N}_L(\mathcal{N}_{L-1}(\ldots\mathcal{N}_1(\boldsymbol{x})))$ and define $J_l := \text{Jacobian}(\mathcal{N}_L)\ldots\text{Jacobian}(\mathcal{N}_{l+1})$ and $\boldsymbol{f}_l := \mathcal{N}_l(\ldots\mathcal{N}_1(\boldsymbol{x}))$. Then the weight matrix $W_l$ at layer $l$ has gradient $G_l$ in the following form for batch size 1:*

***(a)** For $\ell_2$-objective $\varphi := \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{f}_L\|_2^2$:*

$$G_l = \left(J_l^\top \boldsymbol{y} - J_l^\top J_l W_l \boldsymbol{f}_{l-1}\right) \boldsymbol{f}_{l-1}^\top \quad (6)$$

***(b)** Left $P_{\mathbf{1}}^\perp := I - \frac{1}{K}\mathbf{1}\mathbf{1}^\top$ be the zero-mean PSD projection matrix. For $K$-way logsoftmax loss $\varphi(\boldsymbol{y}; \boldsymbol{f}_L) := -\log\left(\frac{\exp(\boldsymbol{y}^\top \boldsymbol{f}_L)}{\mathbf{1}^\top \exp(\boldsymbol{f}_L)}\right)$ with small logits $\|P_{\mathbf{1}}^\perp \boldsymbol{f}_L\|_\infty \ll \sqrt{K}$:*

$$G_l = \left(J_l P_{\mathbf{1}}^\perp \boldsymbol{y} - \gamma K^{-1} J_l^\top P_{\mathbf{1}}^\perp J_l W_l \boldsymbol{f}_{l-1}\right) \boldsymbol{f}_{l-1}^\top \quad (7)$$

*where $\gamma \approx 1$ and $\boldsymbol{y}$ is a data label with $\boldsymbol{y}^\top \mathbf{1} = 1$.*

From the theoretical analysis above, we can see that for batch size $N$, the gradient $G$ has certain structures: $G = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i W C_i)$ for input-dependent matrix $A_i$, Positive Semi-definite (PSD) matrices $B_i$ and $C_i$. In the following, we prove that such a gradient will become low-rank during training in certain conditions:

**Lemma 3.3** (Gradient becomes low-rank during training)**.** *Suppose the gradient follows the parametric form:*

$$G_t = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i W_t C_i) \tag{8}$$

*with constant $A_i$, PSD matrices $B_i$ and $C_i$ after $t \geq t_0$. We study vanilla SGD weight update: $W_t = W_{t-1} + \eta G_{t-1}$. Let $S := \frac{1}{N}\sum_{i=1}^{N} C_i \otimes B_i$ and $\lambda_1 < \lambda_2$ its two smallest distinct eigenvalues. Then the stable rank $\mathrm{sr}(G_t)$ satisfies:*

$$\mathrm{sr}(G_t) \leq \mathrm{sr}(G_{t_0}^{\parallel}) + \left(\frac{1-\eta\lambda_2}{1-\eta\lambda_1}\right)^{2(t-t_0)} \frac{\|G_0 - G_{t_0}^{\parallel}\|_F^2}{\|G_{t_0}^{\parallel}\|_2^2} \tag{9}$$

*where $G_{t_0}^{\parallel}$ is the projection of $G_{t_0}$ onto the minimal eigenspace $\mathcal{V}_1$ of $S$ corresponding to $\lambda_1$.*

In practice, the constant assumption can approximately hold for some time, in which the second term in Eq. 9 goes to zero exponentially and the stable rank of $G_t$ goes down, yielding low-rank gradient $G_t$. The final stable rank is determined by $\mathrm{sr}(G_{t_0}^{\parallel})$, which is estimated to be low-rank by the following:

**Corollary 3.4** (Low-rank $G_t$)**.** *If the gradient takes the parametric form $G_t = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{a}_i - B_i W_t \boldsymbol{f}_i)\boldsymbol{f}_i^\top$ with all $B_i$ full-rank, and $N' := \mathrm{rank}(\{\boldsymbol{f}_i\}) < n$, then $\mathrm{sr}(G_{t_0}^{\parallel}) \leq n - N'$ and thus $\mathrm{sr}(G_t) \leq n/2$ for large $t$.*

**Remarks.** The gradient form is justified by Theorem 3.2. Intuitively, when $N'$ is small, $G_t$ is a summation of $N'$ rank-1 update and is naturally low rank; on the other hand, when $N'$ becomes larger and closer to $n$, then the training dynamics has smaller null space $\mathcal{V}_1$, which also makes $G_t$ low-rank. The full-rank assumption of $\{B_i\}$ is reasonable, e.g., in LLMs, the output dimensions of the networks (i.e., the vocabulary size) is often huge compared to matrix dimensions.

In general if the batch size $N$ is large, then it becomes a bit tricky to characterize the minimal eigenspace $\mathcal{V}_1$ of $S$. On the other hand, if $\mathcal{V}_1$ has nice structure, then $\mathrm{sr}(G_t)$ can be bounded even further:

**Corollary 3.5** (Low-rank $G_t$ with special structure of $\mathcal{V}_1$)**.** *If $\mathcal{V}_1(S)$ is 1-dimensional with decomposable eigenvector $\boldsymbol{v} = \boldsymbol{y} \otimes \boldsymbol{z}$, then $\mathrm{sr}(G_{t_0}^{\parallel}) = 1$ and thus $G_t$ becomes rank-1.*

One rare failure case of Lemma 3.3 is when $G_{t_0}^{\parallel}$ is precisely zero, in which $\mathrm{sr}(G_{t_0}^{\parallel})$ becomes undefined. This happens to be true if $t_0 = 0$, i.e., $A_i$, $B_i$ and $C_i$ are constant throughout the entire training process. Fortunately, for practical training, this does not happen.

**Transformers.** For Transformers, we can also separately prove that the weight gradient of the lower layer (i.e., *project-up*) weight of feed forward network (FFN) becomes low rank over time, using the JoMA framework (**?**). Please check Appendix (Sec. B.3) for details.

### 3.3. Gradient Low-rank Projection (GaLore)

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

**Definition 3.6** (Gradient Low-rank Projection (**GaLore**))**.** Gradient low-rank projection (**GaLore**) denotes the following gradient update rules ($\eta$ is the learning rate):

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t, \quad \tilde{G}_t = P_t \rho_t(P_t^\top G_t Q_t) Q_t^\top \tag{10}$$

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

Different from LoRA, GaLore *explicitly utilizes the low-rank updates* instead of introducing additional low-rank adaptors and hence does not alter the training dynamics.

In the following, we show that GaLore converges under a similar (but more general) form of gradient update rule (Eqn. 8). This form corresponds to Eqn. 6 but with a larger batch size.

**Definition 3.7** ($L$-continuity)**.** A function $\boldsymbol{h}(W)$ has (Lipschitz) $L$-continuity, if for any $W_1$ and $W_2$, $\|\boldsymbol{h}(W_1) - \boldsymbol{h}(W_2)\|_F \leq L\|W_1 - W_2\|_F$.

**Theorem 3.8** (Convergence of GaLore with fixed projections)**.** *Suppose the gradient has the form of Eqn. 8 and $A_i$, $B_i$ and $C_i$ have $L_A$, $L_B$ and $L_C$ continuity with respect to $W$ and $\|W_t\| \leq D$. Let $R_t := P_t^\top G_t Q_t$, $\hat{B}_{it} := P_t^\top B_i(W_t) P_t$, $\hat{C}_{it} := Q_t^\top C_i(W_t) Q_t$ and $\kappa_t := \frac{1}{N}\sum_i \lambda_{\min}(\hat{B}_{it})\lambda_{\min}(\hat{C}_{it})$. If we choose constant $P_t = P$ and $Q_t = Q$, then GaLore with $\rho_t \equiv 1$ satisfies:*

$$\|R_t\|_F \leq \left[1 - \eta(\kappa_{t-1} - L_A - L_B L_C D^2)\right] \|R_{t-1}\|_F \tag{11}$$

*As a result, if $\min_t \kappa_t > L_A + L_B L_C D^2$, $R_t \to 0$ and thus GaLore converges with fixed $P_t$ and $Q_t$.*

**Setting $P$ and $Q$.** The theorem tells that $P$ and $Q$ should project into the subspaces corresponding to the first few

largest eigenvectors of $\hat{B}_{it}$ and $\hat{C}_{it}$ for faster convergence (large $\kappa_t$). While all eigenvalues of the positive semidefinite (PSD) matrix $B$ and $C$ are non-negative, some of them can be very small and hinder convergence (i.e., it takes a long time for $G_t$ to become 0). With the projection $P$ and $Q$, $P^\top B_{it} P$ and $Q^\top C_{it} Q$ only contain the largest eigen subspaces of $B$ and $C$, improving the convergence of $R_t$ and at the same time, reduces the memory usage.

While it is tricky to obtain the eigenstructure of $\hat{B}_{it}$ and $\hat{C}_{it}$ (they are parts of Jacobian), one way is to instead use the spectrum of $G_t$ via Singular Value Decomposition (SVD):

$$G_t = USV^\top \approx \sum_{i=1}^{r} s_i u_i v_i^\top \tag{12}$$

$$P_t = [u_1, u_2, ..., u_r], \quad Q_t = [v_1, v_2, ..., v_r] \tag{13}$$

**Difference between GaLore and LoRA.** While both GaLore and LoRA have "low-rank" in their names, they follow very different training trajectories. For example, when $r = \min(m, n)$, GaLore with $\rho_t \equiv 1$ follows the exact training trajectory of the original model, as $\tilde{G}_t = P_t P_t^\top G_t Q_t Q_t^\top = G_t$. On the other hand, when $BA$ reaches full rank (i.e., $B \in \mathbb{R}^{m \times m}$ and $A \in \mathbb{R}^{m \times n}$), optimizing $B$ and $A$ simultaneously follows a very different training trajectory compared to the original model.

## 4. GaLore for Memory-Efficient Training

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. One reason is that the principal subspaces of $B_t$ and $C_t$ (and thus $G_t$) 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.

### 4.1. Composition of Low-Rank Subspaces

We allow GaLore to switch across low-rank subspaces:

$$W_t = W_0 + \Delta W_{T_1} + \Delta W_{T_2} + \ldots + \Delta W_{T_n}, \tag{14}$$

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 12. We illustrate how the trajectory of $\tilde{G}_t$ traverses through multiple low-rank subspaces in Fig. 2. In the experiment section, we show that allowing multiple low-rank subspaces is the key to achieving the successful pre-training of LLMs.

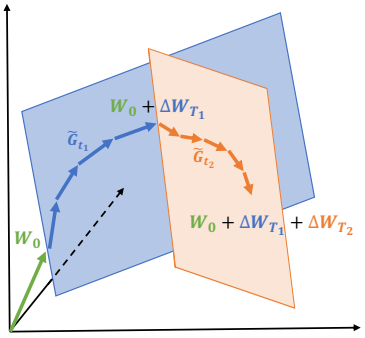


Figure 2: Learning through low-rank subspaces $\Delta W_{T_1}$ and $\Delta W_{T_2}$ using GaLore. For $t_1 \in [0, T_1 - 1]$, $W$ are updated by projected gradients $\tilde{G}_{t_1}$ in a subspace determined by fixed $P_{t_1}$ and $Q_{t_1}$. After $T_1$ steps, the subspace is changed by recomputing $P_{t_2}$ and $Q_{t_2}$ for $t_2 \in [T_1, T_2 - 1]$, and the process repeats until convergence.

Following the above procedure, the switching frequency $T$ becomes a hyperparameter. The ablation study (Fig. 5) shows a sweet spot exists. 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 Theorem 3.8. 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 (convergence proof in Theorem 3.8 only means good progress in the designated subspace, but does not mean good overall performance). While optimal $T$ depends on the total training iterations and task complexity, we find that a value between $T = 50$ to $T = 1000$ makes no much difference. Thus, the total computational overhead induced by SVD is negligible ($< 10\%$) compared to other memory-efficient training techniques such as memory offloading (**?**).

### 4.2. Memory-Efficient Optimization

**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 (**?**). When $\rho_t \equiv \text{Adam}$, by projecting $G_t$ into its low-rank form $R_t$, 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 computes the low-rank normalized gradient $N_t$ as follows:

$$N_t = \rho_t(R_t) = M_t / (\sqrt{V_t} + \epsilon). \tag{15}$$

GaLore can also apply to other optimizers (e.g., Adafactor) that have similar update rules and require a large amount of memory to store gradient statistics.

**Algorithm 2:** Adam with GaLore

**Input:** A layer weight matrix $W \in \mathbb{R}^{m\times n}$ with $m \leq n$. Step size $\eta$, scale factor $\alpha$, decay rates $\beta_1, \beta_2$, 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** $W_t$

**Reducing memory usage of projection matrices.** To achieve the best memory-performance trade-off, we only use one project matrix $P$ or $Q$, projecting the gradient $G$ into $P^\top G$ if $m \leq n$ and $GQ$ otherwise. We present the algorithm applying GaLore to Adam in Algorithm 2.

With this setting, GaLore requires less memory than LoRA during training. As GaLore can always merge $\Delta W_t$ 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.

As Theorem 3.8 does not require the projection matrix to be carefully calibrated, we can further reduce the memory cost of projection matrices by quantization and efficient parameterization, which we leave for future work.

### 4.3. Combining with Existing Techniques

GaLore is compatible with existing memory-efficient optimization techniques. In our work, we mainly consider applying GaLore with 8-bit optimizers and per-layer weight updates.

**8-bit optimizers.** **?** proposed 8-bit Adam optimizer that maintains 32-bit optimizer performance at a fraction of the memory footprint. We apply GaLore directly to the existing implementation of 8-bit Adam.

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 | ✓ | ✓ |

**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 to GaLore, which performs the weight updates during backpropagation. This is the same technique proposed in recent works to reduce memory requirement (**??**).

### 4.4. Hyperparameters of GaLore

In addition to Adam's original hyperparameters, GaLore only introduces very few additional hyperparameters: the rank $r$ which is also present in LoRA, the subspace change frequency $T$ (see Sec. 4.1), and the scale factor $\alpha$.

Scale factor $\alpha$ controls the strength of the low-rank update, which is similar to the scale factor $\alpha/r$ appended to the low-rank adaptor in **?**. We note that the $\alpha$ does not depend on the rank $r$ in our case. This is because, when $r$ is small during pre-training, $\alpha/r$ significantly affects the convergence rate, unlike fine-tuning.

## 5. Experiments

We evaluate GaLore on both pre-training and fine-tuning of LLMs. All experiments run on NVIDIA A100 GPUs.

**Pre-training on C4.** 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 (**?**). 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.

**Architecture and hyperparameters.** We follow the experiment setup from **?**, which adopts a LLaMA-based[3] architecture with RMSNorm and SwiGLU activations (**???**). For each model size, we use the same set of hyperparameters across methods, except the learning rate. We run all

[3]LLaMA materials in our paper are subject to LLaMA community license.

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. 4.

| | **60M** | **130M** | **350M** | **1B** |
|---|---|---|---|---|
| Full-Rank | 34.06 (0.36G) | 25.08 (0.76G) | 18.80 (2.06G) | 15.56 (7.80G) |
| **GaLore** | **34.88** (0.24G) | **25.36** (0.52G) | **18.95** (1.22G) | **15.64** (4.38G) |
| Low-Rank | 78.18 (0.26G) | 45.51 (0.54G) | 37.41 (1.08G) | 142.53 (3.57G) |
| LoRA | 34.99 (0.36G) | 33.92 (0.80G) | 25.58 (1.76G) | 19.21 (6.17G) |
| ReLoRA | 37.04 (0.36G) | 29.37 (0.80G) | 29.08 (1.76G) | 18.33 (6.17G) |
| $r/d_{model}$ | 128 / 256 | 256 / 768 | 256 / 1024 | 512 / 2048 |
| Training Tokens | 1.1B | 2.2B | 6.4B | 13.1B |

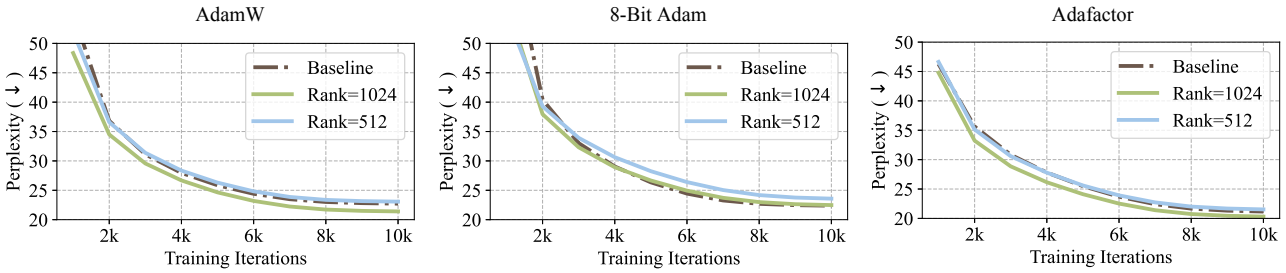


Figure 3: Applying GaLore to different optimizers for pre-training LLaMA 1B on C4 dataset for 10K steps. Validation perplexity over training steps is reported. We apply GaLore to each optimizer with the rank of 512 and 1024, where the 1B model dimension is 2048.

Table 3: Pre-training LLaMA 7B on C4 dataset for 150K steps. Validation perplexity and memory estimate are reported.

| | **Mem** | **40K** | **80K** | **120K** | **150K** |
|---|---|---|---|---|---|
| **8-bit GaLore** | 18G | 17.94 | 15.39 | 14.95 | 14.65 |
| 8-bit Adam | 26G | 18.09 | 15.47 | 14.83 | 14.61 |
| Tokens (B) | | 5.2 | 10.5 | 15.7 | 19.7 |

experiments with BF16 format to reduce memory usage, and we tune the learning rate for each method under the same amount of computational budget and report the best performance. The details of our task setups and hyperparameters are provided in the appendix.

**Fine-tuning on GLUE tasks.** GLUE is a benchmark for evaluating the performance of NLP models on a variety of tasks, including sentiment analysis, question answering, and textual entailment (**?**). We use GLUE tasks to benchmark GaLore against LoRA for memory-efficient fine-tuning.

### 5.1. Comparison with Existing 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: $W = BA$ (**?**).

**LoRA** **?** 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. In the case of pre-training, $W_0$ is the full-rank initialization matrix. We set LoRA alpha to 32 and LoRA dropout to 0.05 as their default settings.

**ReLoRA** **?** proposed ReLoRA, 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. ReLoRA requires careful tuning of merging frequency, learning rate reset, and optimizer states reset. We evaluate ReLoRA without a full-rank training warmup for a fair comparison.

For GaLore, we set subspace frequency $T$ to 200 and scale factor $\alpha$ 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 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. We note that for 1B model size, GaLore even outperforms full-rank baseline when $r = 1024$ instead of $r = 512$. Compared to LoRA and ReLoRA, GaLore requires less memory for storing model parameters and optimizer states. A detailed training setting of each model and memory estimation for each method are in the appendix.

### 5.2. GaLore with Memory-Efficient Optimizers

We demonstrate that GaLore can be applied to various learning algorithms, especially memory-efficient optimizers, to further reduce the memory footprint. We apply GaLore to AdamW, 8-bit Adam, and Adafactor optimizers (**???**). We consider Adafactor with first-order statistics to avoid performance degradation.

We evaluate them on LLaMA 1B architecture with 10K training steps, and we tune the learning rate for each setting and report the best performance. As shown in Fig. 3, applying GaLore does not significantly affect their convergence. By using GaLore with a rank of 512, the memory footprint is reduced by up to 62.5%, on top of the memory savings from using 8-bit Adam or Adafactor optimizer. Since 8-bit Adam requires less memory than others, we denote 8-bit GaLore as GaLore with 8-bit Adam, and use it as the default method for the following experiments on 7B model pre-training and memory measurement.

### 5.3. 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 with 19.7B tokens, using 8-node training in parallel with a total of 64 A100 GPUs. Due to computational constraints, we compare 8-bit GaLore ($r = 1024$) with 8-bit Adam with a single trial without tuning the hyperparameters. As shown in Table 3, after 150K steps, 8-bit GaLore achieves a perplexity of 14.65, comparable to 8-bit Adam with a perplexity of 14.61.

### 5.4. 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. We use hyperparameters from **?** for LoRA and tune the learning rate and scale factor for GaLore. As shown in Table 4, 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.

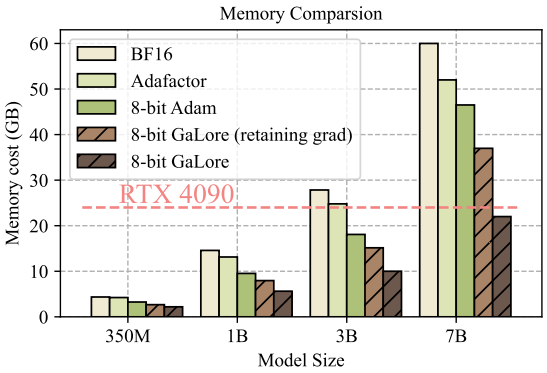


Figure 4: Memory usage for different methods at various model sizes, evaluated with a token batch size of 256. 8-bit GaLore (retaining grad) disables per-layer weight updates but stores weight gradients during training.

### 5.5. Measurement of Memory and Throughput

While Table 2 gives the theoretical benefit of GaLore compared to other methods in terms of memory usage, we also measure the actual memory footprint of training LLaMA models by various methods, with a token batch size of 256. The training is conducted on a single device setup without activation checkpointing, memory offloading, and optimizer states partitioning (**?**).

**Training 7B models on consumer GPUs with 24G memory.** As shown in Fig. 4, 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. In addition, when activation checkpointing is enabled, per-GPU token batch size can be increased up to 4096. 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 (**?**) 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.

**Throughput overhead of GaLore.** We also measure the throughput of the pre-training LLaMA 1B model with 8-bit GaLore and other methods, where the results can be found

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

| | 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 |

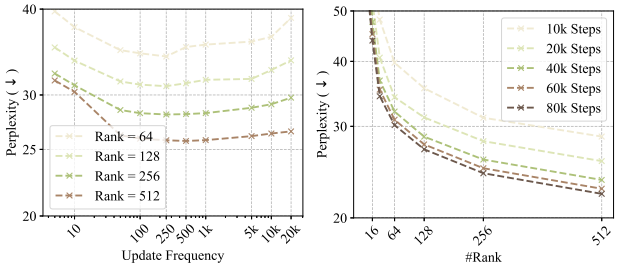


Figure 5: Ablation study of GaLore on 130M models. **Left:** varying subspace update frequency $T$. **Right:** varying subspace rank and training iterations.

in the appendix. Particularly, the current implementation of 8-bit GaLore achieves 1019.63 tokens/second, which induces 17% overhead compared to 8-bit Adam implementation. Disabling per-layer weight updates for GaLore achieves 1109.38 tokens/second, improving the throughput by 8.8%. We note that our results do not require offloading strategies or checkpointing, which can significantly impact training throughput. We leave optimizing the efficiency of GaLore implementation for future work.

## 6. Ablation Study

**How many subspaces are needed during pre-training?** We observe that both too frequent and too slow changes of subspaces hurt the convergence, as shown in Fig. 5 (left). The reason has been discussed in Sec. 4.1. In general, for small $r$, the subspace switching should happen more to avoid wasting optimization steps in the wrong subspace, while for large $r$ the gradient updates cover more subspaces, providing more cushion.

**How does the rank of subspace affect the convergence?** Within a certain range of rank values, decreasing the rank only slightly affects the convergence rate, causing a slowdown with a nearly linear trend. As shown in Fig. 5 (right), training with a rank of 128 using 80K steps achieves a lower loss than training with a rank of 512 using 20K steps. This shows that GaLore can be used to trade-off between memory and computational cost. In a memory-constrained scenario, reducing the rank allows us to stay within the memory budget while training for more steps to preserve the performance.

## 7. 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 identify several open problems for GaLore, which include (1) applying GaLore on training of various models such as vision transformers (**?**) and diffusion models (**?**), (2) further enhancing memory efficiency by employing low-memory projection matrices, and (3) exploring the feasibility of elastic data distributed training on low-bandwidth consumer-grade hardware.

We hope that our work will inspire future research on memory-efficient training from the perspective of gradient low-rank projection. We believe that GaLore will be a valuable tool for the community, enabling the training of large-scale models on consumer-grade hardware with limited resources.

## Impact Statement

This paper aims to improve the memory efficiency of training LLMs in order to reduce the environmental impact of LLM pre-training and fine-tuning. By enabling the training of larger models on hardware with lower memory, our approach helps to minimize energy consumption and carbon footprint associated with training LLMs.

## Acknowledgments

We thank Meta AI for computational support. We appreciate the helpful feedback and discussion from Florian Schäfer, Jeremy Bernstein, and Vladislav Lialin. B. Chen greatly appreciates the support by Moffett AI. Z. Wang is in part supported by NSF Awards 2145346 (CAREER), 02133861 (DMS), 2113904 (CCSS), and the NSF AI Institute for Foundations of Machine Learning (IFML). A.

Anandkumar is supported by the Bren Foundation and the
Schmidt Sciences through AI 2050 senior fellow program.

## A. Additional Related Works

Adafactor (**?**) 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. 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.

The fused backward operation proposed by LOMO (**?**) mitigates the memory cost of storing weight gradients during training. Integrated with the standard SGD optimizer, LOMO achieves zero optimizer and gradient memory cost during training. AdaLOMO (**?**) enhances this approach by combining the fused backward operation with adaptive learning rate for each parameter, similarly achieving minimal optimizer memory cost.

While LOMO and AdaLOMO represent significant advancements in memory-efficient optimization for fine-tuning or continual pre-training, they might not be directly applicable to pre-training from scratch at larger scales. For example, the vanilla Adafactor, adopted by AdaLOMO, has been demonstrated to lead to increased training instabilities at larger scales (**????**). We believe integrating GaLore with the fused backward operation may offer a promising avenue for achieving memory-efficient large-scale pre-training from scratch.

## B. Proofs

### B.1. Reversibility

**Definition B.1** (Reversiblity (**?**))**.** A network $\mathcal{N}$ that maps input $\boldsymbol{x}$ to output $\boldsymbol{y} = \mathcal{N}(\boldsymbol{x})$ is *reversible*, if there exists $L(\boldsymbol{x}; W)$ so that $\boldsymbol{y} = L(\boldsymbol{x}; W)\boldsymbol{x}$, and the backpropagated gradient $\boldsymbol{g_x}$ satisfies $\boldsymbol{g_x} = L^\top(\boldsymbol{x}; W)\boldsymbol{g_y}$, where $\boldsymbol{g_y}$ is the backpropagated gradient at the output $\boldsymbol{y}$. Here $L(\boldsymbol{x}; W)$ depends on the input $\boldsymbol{x}$ and weight $W$ in the network $\mathcal{N}$.

Note that many layers are reversible, including linear layer (without bias), reversible activations (e.g., ReLU, leaky ReLU, polynomials, etc). Furthermore, they can be combined to construct more complicated architectures:

**Property 1.** *If $\mathcal{N}_1$ and $\mathcal{N}_2$ are reversible networks, then (***Parallel***) $\boldsymbol{y} = \alpha_1\mathcal{N}_1(\boldsymbol{x}) + \alpha_2\mathcal{N}_2(\boldsymbol{x})$ is reversible for constants $\alpha_1$ and $\alpha_2$, and (***Composition***) $\boldsymbol{y} = \mathcal{N}_2(\mathcal{N}_1(\boldsymbol{x}))$ is reversible.*

From this property, it is clear that ResNet architecture $\boldsymbol{x} + \mathcal{N}(\boldsymbol{x})$ is reversible, if $\mathcal{N}$ contains bias-free linear layers and reversible activations, which is often the case in practice. For a detailed analysis, please check Appendix A in (**?**). For architectures like self-attention, one possibility is to leverage JoMA (**?**) to analyze, and we leave for future work.

The gradient of chained reversible networks has the following structure:

**Theorem 3.2** (Gradient Form of reversible models)**.** *Consider a chained reversible neural network $\mathcal{N}(\boldsymbol{x}) := \mathcal{N}_L(\mathcal{N}_{L-1}(\ldots\mathcal{N}_1(\boldsymbol{x})))$ and define $J_l := \text{Jacobian}(\mathcal{N}_L)\ldots\text{Jacobian}(\mathcal{N}_{l+1})$ and $\boldsymbol{f}_l := \mathcal{N}_l(\ldots\mathcal{N}_1(\boldsymbol{x}))$. Then the weight matrix $W_l$ at layer $l$ has gradient $G_l$ in the following form for batch size 1:*

***(a)** For $\ell_2$-objective $\varphi := \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{f}_L\|_2^2$:*

$$G_l = \left(J_l^\top \boldsymbol{y} - J_l^\top J_l W_l \boldsymbol{f}_{l-1}\right)\boldsymbol{f}_{l-1}^\top \tag{6}$$

***(b)** Left $P_{\mathbf{1}}^\perp := I - \frac{1}{K}\mathbf{1}\mathbf{1}^\top$ be the zero-mean PSD projection matrix. For $K$-way logsoftmax loss $\varphi(\boldsymbol{y}; \boldsymbol{f}_L) := -\log\left(\frac{\exp(\boldsymbol{y}^\top \boldsymbol{f}_L)}{\mathbf{1}^\top\exp(\boldsymbol{f}_L)}\right)$ with small logits $\|P_{\mathbf{1}}^\perp \boldsymbol{f}_L\|_\infty \ll \sqrt{K}$:*

$$G_l = \left(J_l P_{\mathbf{1}}^\perp \boldsymbol{y} - \gamma K^{-1} J_l^\top P_{\mathbf{1}}^\perp J_l W_l \boldsymbol{f}_{l-1}\right)\boldsymbol{f}_{l-1}^\top \tag{7}$$

*where $\gamma \approx 1$ and $\boldsymbol{y}$ is a data label with $\boldsymbol{y}^\top\mathbf{1} = 1$.*

*Proof.* Note that for layered reversible network, we have

$$\mathcal{N}(\boldsymbol{x}) = \mathcal{N}_L(\mathcal{N}_{L-1}(...\mathcal{N}_1(\boldsymbol{x}))) = K_L(\boldsymbol{x})K_{L-1}(\boldsymbol{x})\ldots K_1(\boldsymbol{x})\boldsymbol{x} \tag{16}$$

Let $\boldsymbol{f}_l := \mathcal{N}_l(\mathcal{N}_{l-1}(\dots\mathcal{N}_1(\boldsymbol{x})))$ and $J_l := K_L(\boldsymbol{x})\dots K_{l+1}(\boldsymbol{x})$, and for linear layer $l$, we can write $\mathcal{N}(\boldsymbol{x}) = J_l W_l \boldsymbol{f}_{l-1}$. Therefore, for the linear layer $l$ with weight matrix $W_l$, we have:

$$\begin{aligned}
\mathrm{d}\varphi &= (\boldsymbol{y} - \mathcal{N}(\boldsymbol{x}))^\top \mathrm{d}\mathcal{N}(\boldsymbol{x}) && (17)\\
&= (\boldsymbol{y} - \mathcal{N}(\boldsymbol{x}))^\top K_L(\boldsymbol{x})\dots K_{l+1}(\boldsymbol{x})\mathrm{d}W_l \boldsymbol{f}_{l-1} \quad + \quad \text{terms not related to } \mathrm{d}W_l && (18)\\
&= (\boldsymbol{y} - J_l W_l \boldsymbol{f}_{l-1})^\top J_l \mathrm{d}W_l \boldsymbol{f}_{l-1} && (19)\\
&= \mathrm{tr}(\mathrm{d}W_l^\top J_l^\top (\boldsymbol{y} - J_l W_l \boldsymbol{f}_{l-1}) \boldsymbol{f}_{l-1}^\top) && (20)
\end{aligned}$$

This gives the gradient of $W_l$:

$$G_l = J_l^\top \boldsymbol{y} \boldsymbol{f}_{l-1}^\top - J_l^\top J_l W_l \boldsymbol{f}_{l-1} \boldsymbol{f}_{l-1}^\top \tag{21}$$

□

**Softmax Case.** Note that for softmax objective with small logits, we can also prove a similar structure of backpropagated gradient, and thus Theorem 3.2 can also apply.

**Lemma B.2** (Gradient structure of softmax loss)**.** *For $K$-way logsoftmax loss $\varphi(\boldsymbol{y}; \boldsymbol{f}) := -\log\left(\frac{\exp(\boldsymbol{y}^\top \boldsymbol{f})}{\mathbf{1}^\top \exp(\boldsymbol{f})}\right)$, let $\hat{\boldsymbol{f}} = P_{\mathbf{1}}^\perp \boldsymbol{f}$ be the zero-mean version of network output $\boldsymbol{f}$, where $P_{\mathbf{1}}^\perp := I - \frac{1}{K}\mathbf{1}\mathbf{1}^\top$, then we have:*

$$-\mathrm{d}\varphi = \boldsymbol{y}^\top \mathrm{d}\hat{\boldsymbol{f}} - \gamma \hat{\boldsymbol{f}}^\top \mathrm{d}\hat{\boldsymbol{f}}/K + O(\hat{\boldsymbol{f}}^2/K)\mathrm{d}\hat{\boldsymbol{f}} \tag{22}$$

*where $\gamma(\boldsymbol{y}, \boldsymbol{f}) \approx 1$ and $\boldsymbol{y}$ is a data label with $\boldsymbol{y}^\top \mathbf{1} = 1$.*

*Proof.* Let $\hat{\boldsymbol{f}} := P_{\mathbf{1}}^\perp \boldsymbol{f}$ be the zero-mean version of network output $\boldsymbol{f}$. Then we have $\mathbf{1}^\top \hat{\boldsymbol{f}} = 0$ and $\boldsymbol{f} = \hat{\boldsymbol{f}} + c\mathbf{1}$. Therefore, we have:

$$-\varphi = \log\left(\frac{\exp(c)\exp(\boldsymbol{y}^\top \hat{\boldsymbol{f}})}{\exp(c)\mathbf{1}^\top \exp(\hat{\boldsymbol{f}})}\right) = \boldsymbol{y}^\top \hat{\boldsymbol{f}} - \log(\mathbf{1}^\top \exp(\hat{\boldsymbol{f}})) \tag{23}$$

Using the Taylor expansion $\exp(x) = 1 + x + \frac{x^2}{2} + o(x^2)$, we have:

$$\mathbf{1}^\top \exp(\hat{\boldsymbol{f}}) = \mathbf{1}^\top(\mathbf{1} + \hat{\boldsymbol{f}} + \frac{1}{2}\hat{\boldsymbol{f}}^2) + o(\hat{\boldsymbol{f}}^2) = K(1 + \hat{\boldsymbol{f}}^\top \hat{\boldsymbol{f}}/2K + o(\hat{\boldsymbol{f}}^2/K)) \tag{24}$$

So

$$-\varphi = \boldsymbol{y}^\top \hat{\boldsymbol{f}} - \log(1 + \hat{\boldsymbol{f}}^\top \hat{\boldsymbol{f}}/2K + o(\hat{\boldsymbol{f}}^2/K)) - \log K \tag{25}$$

Therefore

$$-\mathrm{d}\varphi = \boldsymbol{y}^\top \mathrm{d}\hat{\boldsymbol{f}} - \frac{\gamma}{K}\hat{\boldsymbol{f}}^\top \mathrm{d}\hat{\boldsymbol{f}} + O\left(\frac{\hat{\boldsymbol{f}}^2}{K}\right)\mathrm{d}\hat{\boldsymbol{f}} \tag{26}$$

where $\gamma := (1 + \hat{\boldsymbol{f}}^\top \hat{\boldsymbol{f}}/2K + o(\hat{\boldsymbol{f}}^2/K))^{-1} \approx 1$. □

**Remarks**. With this lemma, it is clear that for a reversible network $\boldsymbol{f} := \mathcal{N}(\boldsymbol{x}) = J_l(\boldsymbol{x}) W_l \boldsymbol{f}_{l-1}(\boldsymbol{x})$, the gradient $G_l$ of $W_l$ has the following form:

$$G_l = \underbrace{J_l P_{\mathbf{1}}^\perp \boldsymbol{y} \boldsymbol{f}_{l-1}}_{A} - \underbrace{\gamma J_l^\top P_{\mathbf{1}}^\perp J_l}_{B} W_l \underbrace{\boldsymbol{f}_{l-1}\boldsymbol{f}_{l-1}^\top/K}_{C} \tag{27}$$

### B.2. Gradient becomes low-rank

**Lemma B.3** (Gradient becomes low-rank during training)**.** *Suppose the gradient follows the parametric form:*

$$G_t = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i W_t C_i) \tag{8}$$

*with constant $A_i$, PSD matrices $B_i$ and $C_i$ after $t \geq t_0$. We study vanilla SGD weight update: $W_t = W_{t-1} + \eta G_{t-1}$. Let $S := \frac{1}{N}\sum_{i=1}^{N} C_i \otimes B_i$ and $\lambda_1 < \lambda_2$ its two smallest distinct eigenvalues. Then the stable rank $\mathrm{sr}(G_t)$ satisfies:*

$$\mathrm{sr}(G_t) \leq \mathrm{sr}(G_{t_0}^{\parallel}) + \left(\frac{1-\eta\lambda_2}{1-\eta\lambda_1}\right)^{2(t-t_0)} \frac{\|G_0 - G_{t_0}^{\parallel}\|_F^2}{\|G_{t_0}^{\parallel}\|_2^2} \tag{9}$$

*where $G_{t_0}^{\parallel}$ is the projection of $G_{t_0}$ onto the minimal eigenspace $\mathcal{V}_1$ of $S$ corresponding to $\lambda_1$.*

*Proof.* We have

$$G_t = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i W_t C_i) = \frac{1}{N}\sum_{i=1}^{N} A_i - B_i(W_{t-1} + \eta G_{t-1})C_i = G_{t-1} - \frac{\eta}{N}\sum_{i=1}^{N} B_i G_{t-1} C_i \tag{28}$$

Let $S := \frac{1}{N}\sum_{i=1}^{N} C_i \otimes B_i$, and $g_t := \mathrm{vec}(G_t) \in \mathbb{R}^{mn}$ be a vectorized version of the gradient $G_t \in \mathbb{R}^{m\times n}$. Using $\mathrm{vec}(BWC) = (C^\top \otimes B)\mathrm{vec}(W)$, we have:

$$g_t = (I - \eta S) g_{t-1} \tag{29}$$

Now let's bound the stable rank of $G_t$:

$$\text{stable-rank}(G_t) := \frac{\|G_t\|_F^2}{\|G_t\|_2^2} \tag{30}$$

Now $\lambda_1 < \lambda_2$ are the smallest two distinct eigenvectors of $S$. The smallest eigenvalue $\lambda_1$ has multiplicity $\kappa_1$. We can decompose $g_0$ into two components, $g_0 = g_0^{\parallel} + g_0^{\perp}$, in which $g_0^{\parallel}$ lies in the $\kappa_1$-dimensional eigenspace $\mathcal{V}_1$ that corresponds to the minimal eigenvalue $\lambda_1$, and $g_0^{\perp}$ is its residue. Then $\mathcal{V}_1 \subset \mathbb{R}^{mn}$ and its orthogonal complements are invariant subspaces under $S$ and thus:

$$\|G_t\|_F^2 = \|g_t\|_2^2 = \|(I-\eta S)^t g_0\|_2^2 = \|(I-\eta S)^t g_0^{\parallel}\|_2^2 + \|(I-\eta S)^t g_0^{\perp}\|_2^2 \tag{31}$$

$$\leq (1-\eta\lambda_2)^{2t}\|g_0^{\perp}\|_2^2 + (1-\eta\lambda_1)^{2t}\|g_0^{\parallel}\|_2^2 \tag{32}$$

On the other hand, by our assumption, $G_0^{\parallel}$ is rank $L$ and thus has SVD decomposition:

$$G_0^{\parallel} = \sum_{l=1}^{L} c_l \boldsymbol{z}_l \boldsymbol{y}_l^\top \tag{33}$$

with orthonormal unit vectors $\{\boldsymbol{z}_l\}_{l=1}^L$ and $\{\boldsymbol{y}_l\}_{l=1}^L$ and singular values $\{c_l\}_{l=1}^L 1$. This means that

$$g_0^{\parallel} = \mathrm{vec}(G_0^{\parallel}) = \sum_{l=1}^{L} c_l(\boldsymbol{y}_l \otimes \boldsymbol{z}_l) =: \sum_{l=1}^{L} c_l \boldsymbol{v}_l \tag{34}$$

with unit vector $\boldsymbol{v}_l := \boldsymbol{y}_l \otimes \boldsymbol{z}_l \in \mathcal{V}_1$. It is clear that

$$\boldsymbol{v}_l^\top \boldsymbol{v}_{l'} = (\boldsymbol{y}_l^\top \otimes \boldsymbol{z}_l^\top)(\boldsymbol{y}_{l'} \otimes \boldsymbol{z}_{l'}) = (\boldsymbol{y}_l^\top \boldsymbol{y}_{l'})(\boldsymbol{z}_l^\top \boldsymbol{z}_{l'}) = \mathbb{I}(l = l') \tag{35}$$

Therefore, by the definition of spectral norm (or matrix 2-norm), we know it corresponds to the largest singular value, which means:

$$\|G_t\|_2 = \max_{\|\boldsymbol{y}'\|_2=1, \|\boldsymbol{z}'\|_2=1} \boldsymbol{z}'^\top G_t \boldsymbol{y}' \tag{36}$$

$$\geq \max_l \boldsymbol{z}_l^\top G_t \boldsymbol{y}_l = \max_l (\boldsymbol{y}_l \otimes \boldsymbol{z}_l)^\top g_t \tag{37}$$

$$= \max_l \boldsymbol{v}_l^\top (1-\eta S)^t g_0 = (1-\eta\lambda_1)^t \max_l \boldsymbol{v}_l^\top g_0 \tag{38}$$

Note that the last equation is because any $\boldsymbol{v} \in \mathcal{V}_1$ is an eigenvector of $S$ with eigenvalue of $\lambda_1$.

Since $\boldsymbol{v}_l^\top g_0 = \boldsymbol{v}_l^\top (g_0^\perp + g_0^\parallel) = c_l$, $\max_l c_l = \|G_0^\parallel\|_2$ and $\|g_0^\parallel\|_2^2 = \|G_0^\parallel\|_F^2$, we have:

$$\text{stable-rank}(G_t) := \frac{\|G_t\|_F^2}{\|G_t\|_2^2} \leq \text{stable-rank}(G_0^\parallel) + \left(\frac{1-\eta\lambda_2}{1-\eta\lambda_1}\right)^{2t} \frac{\|G_0^\perp\|_F^2}{\|G_0^\parallel\|_2^2} \tag{39}$$

□

**Corollary B.4** (Low-rank $G_t$)**.** *If the gradient takes the parametric form $G_t = \frac{1}{N}\sum_{i=1}^N (\boldsymbol{a}_i - B_i W_t \boldsymbol{f}_i)\boldsymbol{f}_i^\top$ with all $B_i$ full-rank, and $N' := \operatorname{rank}(\{\boldsymbol{f}_i\}) < n$, then $\operatorname{sr}(G_{t_0}^\parallel) \leq n - N'$ and thus $\operatorname{sr}(G_t) \leq n/2$ for large $t$.*

*Proof.* Let $C_i = \boldsymbol{f}_i \boldsymbol{f}_i^\top \in \mathbb{R}^{n\times n}$. Since $N' := \operatorname{rank}(\{\boldsymbol{f}_i\}_{i=1}^N) < n$ and $f_i \in \mathbb{R}^n$, the collections of vectors $\{\boldsymbol{f}_i\}_{i=1}^N$ cannot span the entire space $\mathbb{R}^n$. Let $\{\boldsymbol{u}_j\}_{j=1}^{n-N'}$ be the orthonormal bases for the null space of $\{\boldsymbol{f}_i\}_{i=1}^N$, and $\{\boldsymbol{e}_k\}_{k=1}^m$ be any orthonormal bases for $\mathbb{R}^m$. Then the product bases $\{\boldsymbol{u}_j \otimes \boldsymbol{e}_k\}$ form a set of bases for the minimal eigenspace $\mathcal{V}_1$ of $S$ with the minimal eigenvalue of 0. Since $B_i$ are full-rank, no extra dimensions exist for $\mathcal{V}_1$.

Therefore, when we project $G_{t_0}$ onto $\mathcal{V}_1$, we have:

$$G_{t_0}^\parallel = \sum_{j=1}^{n-N'} \sum_{k=1}^m c_{jk} \boldsymbol{u}_j \boldsymbol{e}_k^\top = \sum_{j=1}^{n-N'} \boldsymbol{u}_j \left(\sum_{k=1}^m c_{jk} \boldsymbol{e}_k\right)^\top \tag{40}$$

and thus $\operatorname{sr}(G_{t_0}^\parallel) \leq \operatorname{rank}(G_{t_0}^\parallel) \leq n - N'$, since stable rank is a lower-bound of the rank.

On the other hand, $G_t$ can be written as a summation of $N'$ rank-1 matrices, by representing each $\boldsymbol{f}_i = \sum_{j=1}^{N'} b_{ij} \boldsymbol{f}_j'$ as a linear combination of $\{\boldsymbol{f}_j'\}_{j=1}^{N'}$:

$$G_t = \frac{1}{N}\sum_{i=1}^N (\boldsymbol{a}_i - B_i W_t \boldsymbol{f}_i) \left(\sum_{j=1}^{N'} b_{ij} \boldsymbol{f}_j'\right)^\top = \frac{1}{N}\sum_{j=1}^{N'} \left[\sum_{i=1}^N b_{ij}(\boldsymbol{a}_i - B_i W_t \boldsymbol{f}_i)\right] \boldsymbol{f}_j'^\top \tag{41}$$

and thus has rank at most $N'$. Therefore, when $t$ is sufficiently large so that the second term in Eqn. 39 is negligible, by Lemma 3.3, we have (notice that $N' < n$):

$$\operatorname{sr}(G_t) \leq \min(n - N', N') \leq n/2 \tag{42}$$

□

**Corollary B.5** (Low-rank $G_t$ with special structure of $\mathcal{V}_1$)**.** *If $\mathcal{V}_1(S)$ is 1-dimensional with decomposable eigenvector $\boldsymbol{v} = \boldsymbol{y} \otimes \boldsymbol{z}$, then $\operatorname{sr}(G_{t_0}^\parallel) = 1$ and thus $G_t$ becomes rank-1.*

*Proof.* In this case, we have $g_0^\parallel = \boldsymbol{v}\boldsymbol{v}^\top g_0 \propto \boldsymbol{v}$. Since $\boldsymbol{v} = \boldsymbol{y} \otimes \boldsymbol{z}$, the resulting $G_0^\parallel$ is a rank-1 matrix and thus $\operatorname{sr}(G_{t_0}^\parallel) = 1$. □

### B.3. Gradient Low-rank property for Transformers

Note that Transformers do not belong to the family of reversible networks. However, we can still show that the gradient of the lower layer (i.e., *project-up*) weight $W \in \mathbb{R}^{m\times n}$ of feed forward network (FFN) becomes low rank over time, using the JoMA framework (**?**). Here $m$ is the embedding dimension, and $n$ is the number of hidden nodes in FFNs.

**Lemma B.6** (Gradient of Project-up in Transformer FFNs)**.** *Suppose the embedding matrix $U \in \mathbb{R}^{m\times M}$ is fixed and column-orthonormal ($M$ is vocabulary size), the activation functions are linear and the backpropagated gradient are stationary (**?**), then the training dynamics of transformed project-up matrix $V := U^\top W \in \mathbb{R}^{M\times n}$ satisfies the following:*

$$\dot{V} = \frac{1}{A}\operatorname{diag}\left(\exp\left(\frac{V\circ V}{2}\right)\mathbf{1}\right)\Delta \tag{43}$$

*where $A$ is the normalization factor of softmax, $\circ$ is the Hadamard (element-wise) product and $\Delta$ is defined in the proof. As a result, the gradient of $V$ is "exponentially more low-rank" than $V$ itself.*

*Proof.* Let $\Delta := [\boldsymbol{\Delta}_1, \dots, \boldsymbol{\Delta}_n] \in \mathbb{R}^{M\times n}$, where $\boldsymbol{\Delta}_j := \mathbb{E}_q[g_j \boldsymbol{x}] \in \mathbb{R}^M$. Here $g_j$ is the backpropagated gradient of hidden node $j$ in FFN layer, $\mathbb{E}_q[\cdot]$ is the conditional expectation given the query is token $q$, and $\boldsymbol{x}$ is the representation of token distribution in the previous layer of Transformer. Specifically, for intermediate layer, $\boldsymbol{x}$ represents the activation output of the previous project-up layer; for the first layer, $\boldsymbol{x}$ represents the frequency count of the input tokens. Then following the derivation of Theorem 2 (**?**), we have for each hidden node $j$ and its weight $\boldsymbol{w}_j$, the transformed weight $\boldsymbol{v}_j := U^\top \boldsymbol{w}_j$ satisfies the following dynamics:

$$\dot{\boldsymbol{v}}_j = \frac{1}{A}\boldsymbol{\Delta}_j \circ \exp(\boldsymbol{v}_j^2/2) \tag{44}$$

where $\boldsymbol{v}_j^2 := \boldsymbol{v}_j \circ \boldsymbol{v}_j$ is the element-wise square of a vector and $\circ$ is the Hadamard (element-wise) product. Since $V := [\boldsymbol{v}_1, \dots, \boldsymbol{v}_n]$, Eqn. 43 follows.

Note that the dynamics of $\boldsymbol{v}_j$ shows that the direction of $\boldsymbol{v}_j$ will change over time (because of $\exp(\boldsymbol{v}_j^2/2)$), and it is not clear how such dynamics leads to low-rank $V$ and even more low-rank $\dot{V}$. For this, we per-row decompose the matrix $V$:

$$V := \begin{bmatrix} \boldsymbol{u}_1^\top \\ \boldsymbol{u}_2^\top \\ \dots \\ \boldsymbol{u}_M^\top \end{bmatrix} \tag{45}$$

where $\boldsymbol{u}_l \in \mathbb{R}^n$. We can also do the same for $\Delta$:

$$\Delta := \begin{bmatrix} \boldsymbol{\mu}_1^\top \\ \boldsymbol{\mu}_2^\top \\ \dots \\ \boldsymbol{\mu}_M^\top \end{bmatrix} \tag{46}$$

where $\boldsymbol{\mu}_l \in \mathbb{R}^n$. Then Eqn. 43 can be decomposed along each row:

$$\dot{\boldsymbol{u}}_l = \frac{1}{A}(e^{\boldsymbol{u}_l^2}\cdot \mathbf{1})\boldsymbol{\mu}_l \tag{47}$$

Then it is clear that $\boldsymbol{u}_l$ is always along the direction of $\boldsymbol{\mu}_l$, which is a fixed quality since the backpropagated gradient $g_j$ and input $\boldsymbol{x}$ are assumed to be stationary (and thus $\boldsymbol{\Delta}_j := \mathbb{E}_q[g_j\boldsymbol{x}]$ is a constant).

Therefore, let $\boldsymbol{u}_l(t) = \alpha_l(t)\boldsymbol{\mu}_l$ with initial condition of the magnitude $\alpha_l(0) = 0$, and we have:

$$\dot{\alpha}_l = \frac{1}{A}e^{\alpha_l^2\boldsymbol{\mu}_l^2}\cdot\mathbf{1} = \frac{1}{A}\sum_{j=1}^{n} e^{\alpha_l^2\mu_{lj}^2} \tag{48}$$

where $1 \le l \le M$ is the token index. In the following we will show that for different $l$, the growth of $\alpha_l$ can be very different. This leads to very different row norms of $V$ and $\dot{V}$ over time, leading to their low-rank structures. Note that Eqn. 48 does not have a close form solution, instead we could estimate its growth:

$$\frac{1}{A}e^{\alpha_l^2\bar{\mu}_l^2} \le \dot{\alpha}_l \le \frac{n}{A}e^{\alpha_l^2\bar{\mu}_l^2} \tag{49}$$

where $\bar{\mu}_l^2 := \max_j \mu_{lj}^2$.

Note that both sides have analytic solutions using Gaussian error functions $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t \in [-1, 1]$. Specifically, for dynamic system like $\dot{x} = Ce^{\beta^2 x^2}$, we have

$$e^{-\beta^2 x^2} \mathrm{d}x = C \mathrm{d}t \tag{50}$$

which gives:

$$\frac{\sqrt{\pi}}{2\beta} \mathrm{erf}\left(\beta x(t)\right) = \int_0^{x(t)} e^{-\beta^2 y^2} \mathrm{d}y = Ct \tag{51}$$

or

$$x(t) = \frac{1}{\beta} \mathrm{erf}^{-1}\left(\frac{2\beta C}{\sqrt{\pi}} t\right) \tag{52}$$

For inequality like $\dot{x} \geq Ce^{\beta^2 x^2}$ or $\dot{x} \leq Ce^{\beta^2 x^2}$, similar equation can be derived. Plug that in, we have:

$$\frac{1}{\bar{\mu}_l} \mathrm{erf}^{-1}\left(\frac{2\bar{\mu}_l}{A\sqrt{\pi}} t\right) \leq \alpha_l(t) \leq \frac{1}{\bar{\mu}_l} \mathrm{erf}^{-1}\left(\frac{2n\bar{\mu}_l}{A\sqrt{\pi}} t\right) \tag{53}$$

Let

$$h(t; a) := \frac{1}{a} \mathrm{erf}^{-1}\left(\frac{2}{\sqrt{\pi}} \frac{a}{A} t\right) \tag{54}$$

then $\lim_{t \to A\sqrt{\pi}/2a} h(t; a) = +\infty$, and $h(t; \bar{\mu}_l) \leq \alpha_l(t) \leq nh(t; n\bar{\mu}_l)$.

Let $l^* = \arg\max_l \bar{\mu}_l^*$ be the row with the largest entry of $\mu$, then if $\bar{\mu}_l^* > n\bar{\mu}_l$ for all $l \neq l^*$, then when $t \to t^* := \frac{A\sqrt{\pi}}{2\bar{\mu}_l^*}$, the magnitude $\alpha_{l^*}(t) \geq h(t; \bar{\mu}_{l^*}) \to +\infty$, while $\alpha_l(t) \leq nh(t; n\bar{\mu}_l)$ still stay finite, since its critical time $t' := \frac{A\sqrt{\pi}}{2n\bar{\mu}_l} > t^*$. Since $\alpha_l(t)$ controls the magnitude of each row of $V$, This means that $V$ eventually becomes rank-1 and so does $W$.

Finally, $\dot{V}$ is even more low rank than $V$, since $\dot{\alpha}_l$ has $\alpha_l$ in its exponents. □

### B.4. Convergence of GaLore

**Theorem 3.8** (Convergence of GaLore with fixed projections)**.** *Suppose the gradient has the form of Eqn. 8 and $A_i$, $B_i$ and $C_i$ have $L_A$, $L_B$ and $L_C$ continuity with respect to $W$ and $\|W_t\| \leq D$. Let $R_t := P_t^\top G_t Q_t$, $\hat{B}_{it} := P_t^\top B_i(W_t) P_t$, $\hat{C}_{it} := Q_t^\top C_i(W_t) Q_t$ and $\kappa_t := \frac{1}{N} \sum_i \lambda_{\min}(\hat{B}_{it}) \lambda_{\min}(\hat{C}_{it})$. If we choose constant $P_t = P$ and $Q_t = Q$, then GaLore with $\rho_t \equiv 1$ satisfies:*

$$\|R_t\|_F \leq \left[1 - \eta(\kappa_{t-1} - L_A - L_B L_C D^2)\right] \|R_{t-1}\|_F \tag{11}$$

*As a result, if $\min_t \kappa_t > L_A + L_B L_C D^2$, $R_t \to 0$ and thus GaLore converges with fixed $P_t$ and $Q_t$.*

*Proof.* Using $\mathrm{vec}(AXB) = (B^\top \otimes A)\mathrm{vec}(X)$ where $\otimes$ is the Kronecker product, the gradient assumption can be written as the following:

$$g_t = a_t - S_t w_t \tag{55}$$

where $g_t := \mathrm{vec}(G_t) \in \mathbb{R}^{mn}$, $w_t := \mathrm{vec}(W_t) \in \mathbb{R}^{mn}$ be the vectorized versions of $G_t$ and $W_t$, $a_t := \frac{1}{N} \sum_i \mathrm{vec}(A_{it})$ and $S_t = \frac{1}{N} \sum_i C_{it} \otimes B_{it}$ are $mn$-by-$mn$ PSD matrix.

Using the same notation, it is clear to show that:

$$(Q \otimes P)^\top g_t = (Q^\top \otimes P^\top)\mathrm{vec}(G_t) = \mathrm{vec}(P^\top G_t Q) = \mathrm{vec}(R_t) =: r_t \tag{56}$$

$$\tilde{g}_t := \mathrm{vec}(\tilde{G}_t) = \mathrm{vec}(PP^\top G_t Q Q^\top) = (Q \otimes P)\mathrm{vec}(R_t) = (Q \otimes P) r_t \tag{57}$$

Then we derive the recursive update rule for $g_t$:

$$\begin{aligned} g_t &= a_t - S_t w_t && (58)\\ &= (a_t - a_{t-1}) + (S_{t-1} - S_t)w_t + a_{t-1} - S_{t-1}w_t && (59)\\ &= e_t + a_{t-1} - S_{t-1}(w_{t-1} + \eta\tilde{g}_{t-1}) && (60)\\ &= e_t + g_{t-1} - \eta S_{t-1}\tilde{g}_{t-1} && (61) \end{aligned}$$

where $e_t := (a_t - a_{t-1}) + (S_{t-1} - S_t)w_t$. Left multiplying by $(Q \otimes P)^\top$, we have:

$$r_t = (Q \otimes P)^\top e_t + r_{t-1} - \eta(Q \otimes P)^\top S_{t-1}(Q \otimes P)r_{t-1} \tag{62}$$

Let

$$\hat{S}_t := (Q \otimes P)^\top S_t(Q \otimes P) = \frac{1}{N}\sum_i (Q \otimes P)^\top (C_{it} \otimes B_{it})(Q \otimes P) = \frac{1}{N}\sum_i (Q^\top C_{it} Q) \otimes (P^\top B_{it} P) \tag{63}$$

Then we have:

$$r_t = (I - \eta\hat{S}_{t-1})r_{t-1} + (Q \otimes P)^\top e_t \tag{64}$$

Now we bound the norm. Note that since $P$ and $Q$ are projection matrices with $P^\top P = I$ and $Q^\top Q = I$, we have:

$$\|(Q \otimes P)^\top e_t\|_2 = \|\mathrm{vec}(P^\top E_t Q)\|_2 = \|P^\top E_t Q\|_F \le \|E_t\|_F \tag{65}$$

where $E_t := \frac{1}{N}\sum_i (A_{it} - A_{i,t-1}) + \frac{1}{N}\sum_i (B_{i,t-1}W_t C_{i,t-1} - B_{it}W_t C_{it})$. So we only need to bound $\|E_t\|_F$. Note that:

$$\begin{aligned} \|A_t - A_{t-1}\|_F &\le L_A\|W_t - W_{t-1}\|_F = \eta L_A\|\tilde{G}_{t-1}\|_F \le \eta L_A\|R_{t-1}\|_F && (66)\\ \|(B_t - B_{t-1})W_t C_{t-1}\|_F &\le L_B\|W_t - W_{t-1}\|_F\|W_t\|_F\|C_{t-1}\|_F = \eta L_B L_C D^2\|R_{t-1}\|_F && (67)\\ \|B_t W_t(C_{t-1} - C_t)\|_F &\le L_C\|B_t\|_F\|W_t\|_F\|W_{t-1} - W_t\|_F = \eta L_B L_C D^2\|R_{t-1}\|_F && (68) \end{aligned}$$

Now we estimate the minimal eigenvalue of $\hat{S}_{t-1}$. Let $\underline{\lambda}_{it} := \lambda_{\min}(P^\top B_{it} P)$ and $\underline{\nu}_{it} := \lambda_{\min}(Q^\top C_{it} Q)$, then $\lambda_{\min}((P^\top B_{it}P) \otimes (Q^\top C_{it}Q)) = \underline{\lambda}_{it}\underline{\nu}_{it}$ and for any unit vector $\boldsymbol{v}$:

$$\boldsymbol{v}^\top \hat{S}_t \boldsymbol{v} = \frac{1}{N}\sum_i \boldsymbol{v}^\top \left[(P^\top B_{it}P) \otimes (Q^\top C_{it}Q)\right] \boldsymbol{v} \ge \frac{1}{N}\sum_i \underline{\lambda}_{it}\underline{\nu}_{it} \tag{69}$$

And thus $\lambda_{\min}(\hat{S}_t) \ge \frac{1}{N}\sum_i \underline{\lambda}_{it}\underline{\nu}_{it}$. Therefore, $\lambda_{\max}(I - \eta\hat{S}_{t-1}) \le 1 - \frac{\eta}{N}\sum_i \underline{\lambda}_{i,t-1}\underline{\nu}_{i,t-1}$. Therefore, let $\kappa_t := \frac{1}{N}\sum_i \underline{\lambda}_{it}\underline{\nu}_{it}$ and using the fact that $\|r_t\|_2 = \|R_t\|_F$, we have:

$$\|R_t\|_F \le \left[1 - \eta(\kappa_{t-1} - L_A - 2L_B L_C D^2)\right]\|R_{t-1}\|_F \tag{70}$$

and the conclusion follows. □

## 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 5 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.

Table 5: 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 | 150K | 19.7 B |

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 $\alpha$ of $0.25$, and the subspace change frequency of $T$ of 200. We note that since $\alpha$ 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 $\leq 0.001$ to avoid spikes in the training loss.

### 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 $42.7$M parameters on low-rank adaptors and $60M$ parameters on the original weights, resulting in a memory cost of $0.20$G for weight parameters and $0.17$G for optimizer states. Table 6 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 6: 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 | 1.19G |
| LoRA | 0.20G | 0.44G | 1.04G | 3.79G |
| ReLoRA | 0.20G | 0.44G | 1.04G | 3.79G |

(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 | 1.78G |
| Low-Rank | 0.17G | 0.37G | 0.72G | 2.38G |
| LoRA | 0.17G | 0.37G | 0.72G | 2.38G |
| ReLoRA | 0.17G | 0.37G | 0.72G | 2.38G |

### C.3. Training Progression

We show the training progression of 130M, 350M, 1B and 7B models in Figure 6. Compared to LoRA, GaLore closely matches the training trajectory of the full-rank baseline, and it even converges slightly faster at the beginning of the training.

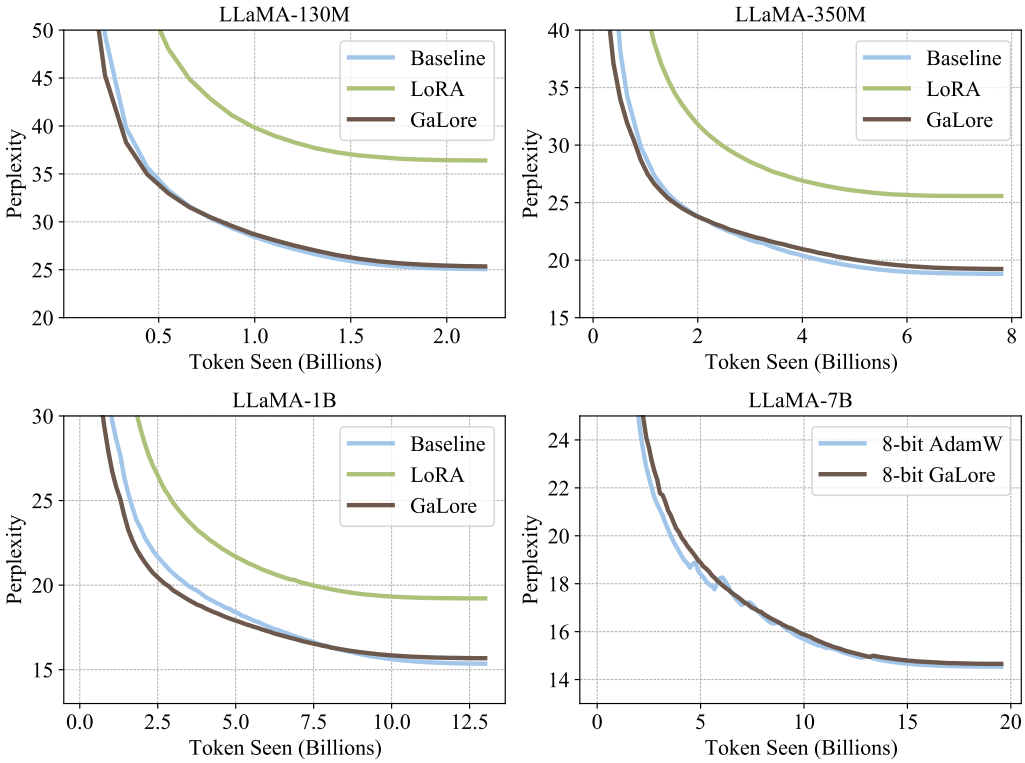


Figure 6: Training progression for pre-training LLaMA models on C4 dataset.

## D. Fine-Tuning Experiments

### D.1. Details of Fine-Tuning on GLUE

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 tune the learning rate and scale factor for GaLore. Table 7 shows the hyperparameters used for fine-tuning RoBERTa-Base for GaLore.

Table 7: 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 $\alpha$ | | | | 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 $\alpha$ | | | | 2 | | | | |
| Max Seq. Len. | | | | 512 | | | | |

### D.2. Fine-Tuning on SQuAD dataset

We evaluate GaLore on the SQuAD dataset (**?**) using the pre-trained BERT-Base model. We use rank 16 for both GaLore and LoRA. GaLore outperforms LoRA in both Exact Match and F1 scores.

Table 8: Evaluating GaLore on SQuAD dataset. Both Exact Match and F1 scores are reported.

| | **Exact Match** | **F1** |
|---|---|---|
| Baseline | 80.83 | 88.41 |
| **GaLore** | **80.52** | **88.29** |
| LoRA | 77.99 | 86.11 |

### D.3. Fine-Tuning on OpenAssistant Conversations Dataset

We apply GaLore on fine-tuning experiments on the OpenAssistant Conversations dataset (**?**), using the pre-trained models, including Gemma-2b, Phi-2, and LLaMA-7B (**??**). We use rank of 128 for both GaLore and LoRA. The results are shown in Table 9.

### D.4. Fine-Tuning on Belle-1M Dataset

We also apply GaLore on fine-tuning experiments on the Belle-1M dataset (**?**), using the pre-trained models, including Gemma-2b, Phi-2, and LLaMA-7B. We use rank of 128 for both GaLore and LoRA. The results are shown in Table 10.

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

Table 9: Evaluating GaLore on OpenAssistant Conversations dataset. Testing perplexity is reported.

| | **Gemma-2b** | **Phi-2** | **LLaMA-7B** |
|---|---|---|---|
| Baseline | 4.53 | 3.81 | 2.98 |
| **GaLore** | **4.51** | **3.83** | 2.95 |
| LoRA | 4.56 | 4.24 | **2.94** |

Table 10: Evaluating GaLore on Belle-1M dataset. Testing perplexity is reported.

| | **Gemma-2b** | **Phi-2** | **LLaMA-7B** |
|---|---|---|---|
| Baseline | 5.44 | 2.66 | 2.27 |
| **GaLore** | **5.35** | **2.62** | **2.28** |
| LoRA | 5.37 | 2.75 | 2.30 |

## 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 11.

Table 11: 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 |