Natural GaLore: Accelerating GaLore for memory-efficient LLM Training and Fine-tuning

Anonymous authors
Paper under double-blind review

Abstract

Training large language models (LLMs) presents significant memory challenges due to the growing size of data, weights, and optimizer states. While techniques such as data and model parallelism, gradient checkpointing, and offloading strategies address this issue, they are often infeasible due to hardware constraints. Alternative methods, like Parameter Efficient Fine-Tuning (PEFT) and GaLore, mitigate memory usage by approximating weights or optimizer states. PEFT methods, such as LoRA and ReLoRa, have gained popularity for fine-tuning LLMs, though they require a full-rank warm start. In contrast, GaLore allows full-parameter learning while being more memory-efficient. In this work, we introduce **Natural GaLore**, which efficiently applies the inverse Empirical Fisher Information Matrix to low-rank gradients using the Woodbury Identity. We show that especially in the case of a limited iteration budget, incorporating second-order information can significantly improve the convergence rate. Empirical pre-training on 60M, 300M, and 1.1B parameter Llama models on C4 data demonstrates significantly lower perplexity over GaLore, with no memory overhead. Furthermore, fine-tuning the TinyLlama 1.1B model for function calling using the TinyAgent framework shows that **Natural GaLore** significantly outperforms LoRA, achieving 83.5% accuracy and surpasses ChatGPT4o with 79.08% on the TinyAgent dataset, all while using 30% less memory.

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.

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

[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 (**?**).

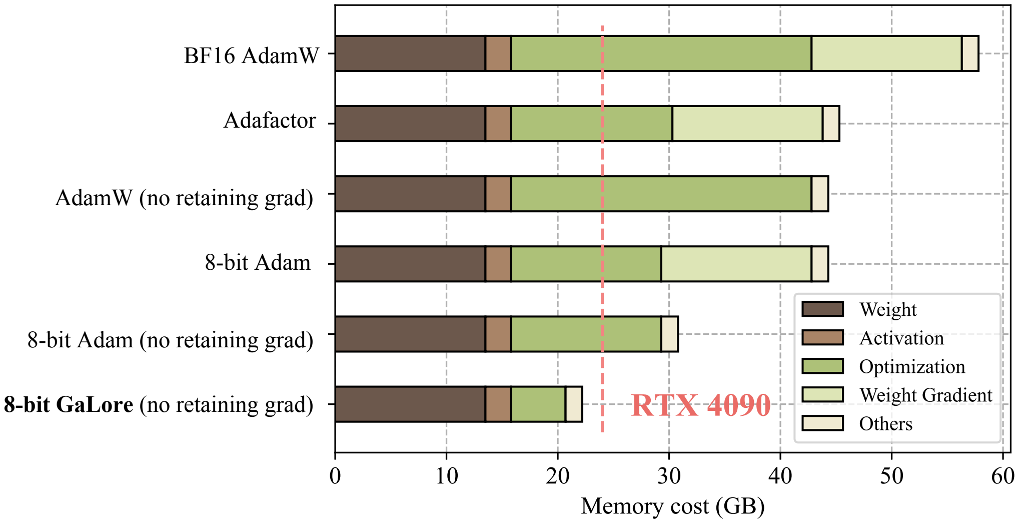

Figure 1: Estimated memory consumption of pre-training a LLaMA 7B model with a token batch size of 256 on a single device, without activation checkpointing and memory offloading[2]. Details refer to Section **??**.

Algorithm 1: Natural 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
```

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 (**Natural 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 Natural 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 $P^\top G Q$. 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. Natural GaLore is more memory-efficient than LoRA as shown in Table **??**. In practice, this yields up to 30% memory reduction compared to LoRA during pre-training.

We demonstrate that Natural GaLore works well in both LLM pre-training and fine-tuning. When pre-training LLaMA 7B on C4 dataset, 8-bit Natural 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, Natural GaLore keeps low memory throughout the entire training, without requiring full-rank training warmup like ReLoRA. Thanks to Natural 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. **??**).

Natural 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, Natural GaLore achieves an average score of 85.89, outperforming LoRA, which achieves a score of 85.61.

As a gradient projection method, Natural 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 **??**. Our experiment (Fig. **??**) 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 Natural 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. Natural GaLore is closely related to the traditional topic of projected gradient descent (PGD) (**??**). A key difference is that, Natural 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 **??**, Theorem **??**, and Theorem **??**). 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 Natural 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 (η is the learning rate):

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

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

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

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

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

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

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:

$$W_T = W_0 + B_T A_T, \tag{5}$$

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 $\boldsymbol{g}_{\boldsymbol{x}}$ satisfies $\boldsymbol{g}_{\boldsymbol{x}} = L^\top(\boldsymbol{x}; W)\boldsymbol{g}_{\boldsymbol{y}}$, where $\boldsymbol{g}_{\boldsymbol{y}}$ 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 **??** 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 $\mathcal{N}(\boldsymbol{x}) := \mathcal{N}_L(\mathcal{N}_{L-1}(\dots\mathcal{N}_1(\boldsymbol{x})))$ and define $J_l := \mathrm{Jacobian}(\mathcal{N}_L)\dots\mathrm{Jacobian}(\mathcal{N}_{l+1})$ and $\boldsymbol{f}_l := \mathcal{N}_l(\dots\mathcal{N}_1(\boldsymbol{x}))$. Then the weight matrix W_l at layer l has gradient G_l in the following form for batch size 1:*

***(a)** For ℓ_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 $\boldsymbol{y}^\top \mathbf{1} = 1$.

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:

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

where $G_{t_0}^{\|}$ is the projection of G_{t_0} onto the minimal eigenspace $\mathcal{V}_1$ of S corresponding to λ_1.

In practice, the constant assumption can approximately hold for some time, in which the second term in Eq. **??** 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 $\mathrm{sr}(G_{t_0}^{\|})$, 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 $\mathrm{sr}(G_{t_0}^{\|}) \leq n - N'$ and thus $\mathrm{sr}(G_t) \leq n/2$ for large t.*

Remarks. The gradient form is justified by Theorem **??**. 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 $\mathrm{sr}(G_{t_0}^{\|}) = 1$ and thus G_t becomes rank-1.*

One rare failure case of Lemma **??** is when $G_{t_0}^{\|}$ is precisely zero, in which $\mathrm{sr}(G_{t_0}^{\|})$ 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. **??**) for details.

3.3 GRADIENT LOW-RANK PROJECTION (NATURAL 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 Natural GaLore strategy:

Definition 3.6 (Gradient Low-rank Projection (**Natural GaLore**))**.** Gradient low-rank projection (**Natural GaLore**) denotes the following gradient update rules (η 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, Natural 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 Natural GaLore converges under a similar (but more general) form of gradient update rule (Eqn. **??**). This form corresponds to Eqn. **??** 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 Natural GaLore with fixed projections)**.** *Suppose the gradient has the form of Eqn.* **??** *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 Natural 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 Natural 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 κ_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 Natural GaLore and LoRA. While both Natural GaLore and LoRA have "low-rank" in their names, they follow very different training trajectories. For example, when $r = \min(m, n)$, Natural 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 Natural 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, Natural 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 Natural 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 **??**. We illustrate

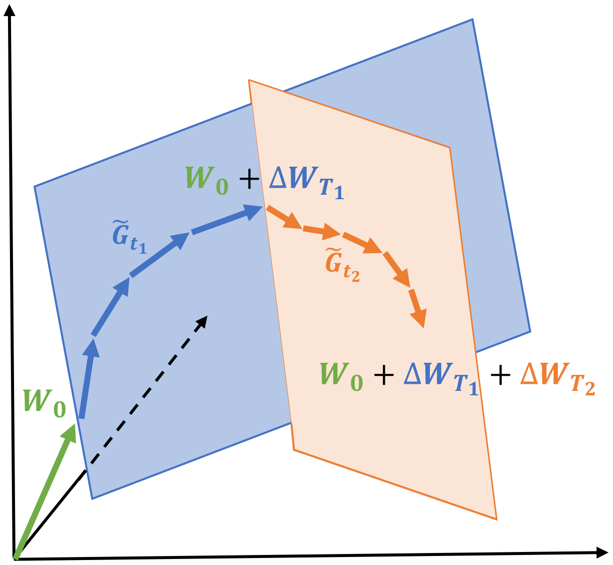

Figure 2: Learning through low-rank subspaces ΔW_{T_1} and ΔW_{T_2} using Natural GaLore. For $t_1 \in [0, T_1 - 1]$, W are updated by projected gradients $\tilde{G}_{t_1}$ in a subspace determined by fixed P_{t_1} and Q_{t_1}. After T_1 steps, the subspace is changed by recomputing P_{t_2} and Q_{t_2}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

Algorithm 2: Adam with Natural 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 $t \leftarrow 0$ $G_t \in \mathbb{R}^{m \times n} \leftarrow -\nabla_W \varphi_t(W_t)$ $t \bmod T = 0$ $U, S, V \leftarrow \text{SVD}(G_t)$
$P_t \leftarrow U[:, :r]$   Initialize left projector as $m \leq n$ $P_t \leftarrow P_{t-1}$   Reuse the previous projector $R_t \leftarrow P_t^\top G_t$   Project
gradient into compact space________________UPDATE$(R_t)$ **by Adam** .5em   $M_t \leftarrow \beta_1 \cdot M_{t-1} + (1 - \beta_1) \cdot R_t$
$V_t \leftarrow \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot R_t^2$   $M_t \leftarrow M_t / (1 - \beta_1^t)$   $V_t \leftarrow V_t / (1 - \beta_2^t)$   $N_t \leftarrow M_t / (\sqrt{V_t} + \epsilon)$
$\tilde{G}_t \leftarrow \alpha \cdot P N_t$         Project back to original space $W_t \leftarrow W_{t-1} + \eta \cdot \tilde{G}_t$ $t \leftarrow t + 1$ convergence criteria met $W_t$
```

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

Following the above procedure, the switching frequency T becomes a hyperparameter. The ablation study (Fig. **??**) 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 **??**. 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 **??** 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. Natural GaLore significantly reduces the memory cost of optimizer that heavily rely on component-wise gradient statistics, such as Adam (**?**). When $\rho_t \equiv \mathrm{Adam}$, 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. Natural GaLore computes the low-rank normalized gradient N_t as

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

2:

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

follows:

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

Natural 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.

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 Natural GaLore to Adam in Algorithm **??**.

With this setting, Natural GaLore requires less memory than LoRA during training. As Natural GaLore can always merge ΔW_t to W_0 during weight updates, it does not need to store a separate low-rank factorization BA. In total, Natural GaLore requires $(mn + mr + 2nr)$ memory, while LoRA requires $(mn + 3mr + 3nr)$ memory. A comparison between Natural GaLore and LoRA is shown in Table **??**.

As Theorem **??** 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

Natural GaLore is compatible with existing memory-efficient optimization techniques. In our work, we mainly consider applying Natural 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 Natural GaLore directly to the existing implementation of 8-bit Adam.

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 Natural 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 Natural GaLore

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

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 adaptor in **?**. We note that the α does not depend on the rank r in our case. This is because, when r is small during pre-training, α/r significantly affects the convergence rate, unlike fine-tuning.

4.5 Introduction

Natural gradient methods are optimization algorithms that adjust parameter updates according to the geometry of the parameter space, leading to faster convergence compared to standard gradient descent. They precondition the gradient using the inverse of the Fisher Information Matrix (FIM). However, computing and storing the full FIM and its inverse is computationally infeasible for large-scale models due to their high dimensionality.

To address this challenge, we propose an **online natural gradient algorithm** that operates in a low-rank subspace of the gradient space. By projecting gradients onto this subspace and approximating the FIM within it, we can efficiently compute natural gradient updates without explicit layer-wise information.

4.6 Algorithm Description

The algorithm consists of the following key steps:

1. **Low-Rank Gradient Projection**
2. **Gradient History Buffer Maintenance**
3. **FIM Approximation Using Gradient History**
4. **Natural Gradient Computation via Woodbury Identity**
5. **Efficient Computation Using Cholesky Decomposition**

We will explain each step in detail.

4.6.1 Step 1: Low-Rank Gradient Projection

Given a full-rank gradient tensor $\boldsymbol{g} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, we project it onto a low-rank subspace using Tucker decomposition. The Tucker decomposition approximates $\boldsymbol{g}$ as:

$$\boldsymbol{g} \approx \mathcal{G} = \mathcal{C} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \cdots \times_d \mathbf{U}^{(d)},$$

where:

- $\mathcal{C} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_d}$ is the core tensor.
- $\mathbf{U}^{(i)} \in \mathbb{R}^{n_i \times r_i}$ are the factor matrices (orthogonal matrices).
- $\times_i$ denotes the mode-i tensor-matrix product.
- r_i is the rank along mode i, with $r_i \ll n_i$.

The low-rank representation (transformed gradient) is obtained by:

$$\boldsymbol{g}_{\text{low}} = \text{Transform}(\boldsymbol{g}) = \boldsymbol{g} \times_1 (\mathbf{U}^{(1)})^\top \times_2 (\mathbf{U}^{(2)})^\top \times_3 \cdots \times_d (\mathbf{U}^{(d)})^\top.$$

4.6.2 Step 2: Gradient History Buffer Maintenance

We maintain a buffer of the recent s low-rank gradient vectors to capture local curvature information. Let $\boldsymbol{g}_t$ be the transformed low-rank gradient at iteration t, flattened into a vector:

$$\boldsymbol{g}_t \in \mathbb{R}^k, \quad \text{where } k = r_1 r_2 \cdots r_d.$$

We store the recent s gradients in the matrix $\mathbf{G}$:

$$\mathbf{G} = [\boldsymbol{g}_{t-s+1}, \boldsymbol{g}_{t-s+2}, \ldots, \boldsymbol{g}_t] \in \mathbb{R}^{k \times s}.$$

4.6.3 Step 3: Approximating the Fisher Information Matrix

We approximate the FIM within the low-rank subspace using the outer products of the stored gradients:

$$\mathbf{F} = \lambda \mathbf{I}_k + \mathbf{G}\mathbf{G}^\top,$$

where:

- $\lambda > 0$ is a damping term to ensure numerical stability.
- $\mathbf{I}_k$ is the $k \times k$ identity matrix.

4.6.4 Step 4: Computing the Natural Gradient via Woodbury Identity

Directly computing $\mathbf{F}^{-1}$ is computationally expensive for large k. Instead, we use the **Woodbury identity** to efficiently compute $\mathbf{F}^{-1}\boldsymbol{g}_t$:

$$\mathbf{F}^{-1}\boldsymbol{g}_t = \frac{1}{\lambda}\left(\boldsymbol{g}_t - \mathbf{G}\left(\lambda\mathbf{I}_s + \mathbf{G}^\top\mathbf{G}\right)^{-1}\mathbf{G}^\top\boldsymbol{g}_t\right).$$

Let us define:

- $\mathbf{S} = \lambda\mathbf{I}_s + \mathbf{G}^\top\mathbf{G} \in \mathbb{R}^{s\times s}$.
- $\boldsymbol{y} = \mathbf{G}^\top\boldsymbol{g}_t \in \mathbb{R}^s$.

Then, we compute:

1. $\mathbf{S} = \lambda\mathbf{I}_s + \mathbf{G}^\top\mathbf{G}$.
2. Solve $\mathbf{S}\boldsymbol{z} = \boldsymbol{y}$ for $\boldsymbol{z}$.
3. Compute $\mathbf{F}^{-1}\boldsymbol{g}_t = \frac{1}{\lambda}(\boldsymbol{g}_t - \mathbf{G}\boldsymbol{z})$.

4.6.5 Step 5: Efficient Computation Using Cholesky Decomposition

To efficiently solve the linear system $\mathbf{S}\boldsymbol{z} = \boldsymbol{y}$, we use **Cholesky decomposition**:

1. Compute the Cholesky factorization of $\mathbf{S}$:
$$\mathbf{S} = \mathbf{L}\mathbf{L}^\top,$$
where $\mathbf{L}$ is a lower triangular matrix.
2. Solve for $\boldsymbol{u}$ in the forward substitution:
$$\mathbf{L}\boldsymbol{u} = \boldsymbol{y}.$$
3. Solve for $\boldsymbol{z}$ in the backward substitution:
$$\mathbf{L}^\top\boldsymbol{z} = \boldsymbol{u}.$$

4.7 Algorithm Summary

At each iteration t, the algorithm proceeds as follows:

1. **Project the Full-Rank Gradient**:
   - Obtain the low-rank transformed gradient $\boldsymbol{g}_t$ using the current factors $\{\mathbf{U}^{(i)}\}$.
2. **Update Gradient History**:
   - If the history buffer is full, remove the oldest gradient.
   - Add $\boldsymbol{g}_t$ to the history buffer $\mathbf{G}$.
3. **Compute Intermediate Quantities**:
   - $\mathbf{S} = \lambda\mathbf{I}_s + \mathbf{G}^\top\mathbf{G}$.
   - $\boldsymbol{y} = \mathbf{G}^\top\boldsymbol{g}_t$.
4. **Solve for $\boldsymbol{z}$**:
   - Use Cholesky decomposition of $\mathbf{S}$ to solve $\mathbf{S}\boldsymbol{z} = \boldsymbol{y}$.
5. **Compute the Natural Gradient**:
   - $\tilde{\boldsymbol{g}}_t = \frac{1}{\lambda}(\boldsymbol{g}_t - \mathbf{G}\boldsymbol{z})$.
6. **Reshape and Project Back**:
   - Reshape $\tilde{\boldsymbol{g}}_t$ back to the low-rank tensor shape.
   - Optionally, project back to the full parameter space using the inverse transform.

4.8 Mathematical Formulas

4.8.1 Low-Rank Projection

The transformed low-rank gradient is obtained by:

$$\boldsymbol{g}_{\text{low}} = \boldsymbol{g} \times_1 (\mathbf{U}^{(1)})^\top \times_2 (\mathbf{U}^{(2)})^\top \times_3 \cdots \times_d (\mathbf{U}^{(d)})^\top.$$

4.8.2 FIM Approximation

We approximate the FIM as:

$$\mathbf{F} = \lambda \mathbf{I}_k + \mathbf{G}\mathbf{G}^\top.$$

4.8.3 Natural Gradient Computation

Compute the natural gradient using:

$$\tilde{\boldsymbol{g}}_t = \mathbf{F}^{-1}\boldsymbol{g}_t = \frac{1}{\lambda}\left(\boldsymbol{g}_t - \mathbf{G}\boldsymbol{z}\right),$$

where $\boldsymbol{z}$ solves:

$$\left(\lambda \mathbf{I}_s + \mathbf{G}^\top \mathbf{G}\right)\boldsymbol{z} = \mathbf{G}^\top \boldsymbol{g}_t.$$

4.8.4 Efficient Solution via Cholesky Decomposition

1. Compute $\mathbf{L}$ such that:

$$\lambda \mathbf{I}_s + \mathbf{G}^\top \mathbf{G} = \mathbf{L}\mathbf{L}^\top.$$

2. Solve for $\boldsymbol{u}$:

$$\mathbf{L}\boldsymbol{u} = \boldsymbol{y}.$$

3. Solve for $\boldsymbol{z}$:

$$\mathbf{L}^\top \boldsymbol{z} = \boldsymbol{u}.$$

4. Compute the natural gradient:

$$\tilde{\boldsymbol{g}}_t = \frac{1}{\lambda}\left(\boldsymbol{g}_t - \mathbf{G}\boldsymbol{z}\right).$$

4.9 Notes on Implementation

4.9.1 Avoiding Large Matrix Inversions

By using the Woodbury identity and Cholesky decomposition, we avoid inverting large matrices of size $k \times k$, where k can be very large.

4.9.2 Computational Efficiency

- **Matrix-Vector Products**: Preferred over matrix-matrix multiplications for efficiency on GPUs.
- **GPU Computation**: All tensors are kept on the GPU to avoid data transfer overhead.
- **Efficient Linear Algebra**: Utilizing optimized GPU routines for operations like matrix multiplication and Cholesky decomposition.

4.9.3 Memory Efficiency

The history size s is kept small (e.g., $s = 10$) to limit memory usage and computational cost.

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 Natural GaLore is reported in Fig. **??**.

	60M	**130M**	**350M**	**1B**
Full-Rank	34.06 (0.36G)	25.08 (0.76G)	18.80 (2.06G)	15.56 (7.80G)
Natural 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

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 Natural 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

4.9.4 Numerical Stability

The damping term λ ensures that $\mathbf{S}$ is positive definite, allowing for stable Cholesky decomposition.

4.10 Conclusion

The proposed online natural gradient algorithm efficiently approximates the natural gradient in a low-rank subspace without requiring explicit layer-wise information. By maintaining a history of recent low-rank gradients and utilizing the Woodbury identity along with Cholesky decomposition, we can compute natural gradient updates efficiently on GPUs. This makes the method suitable for large-scale optimization problems where full natural gradient methods are computationally prohibitive.

5 Experiments

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

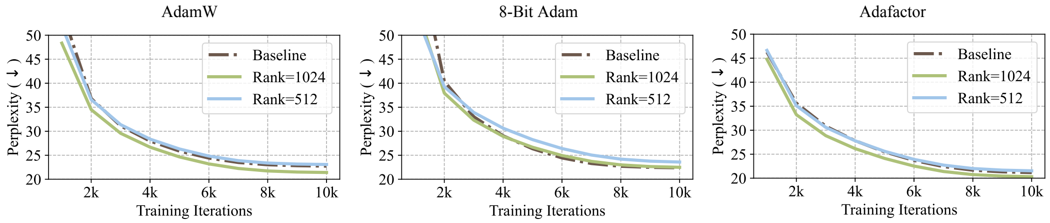

Figure 3: Applying Natural GaLore to different optimizers for pre-training LLaMA 1B on C4 dataset for 10K steps. Validation perplexity over training steps is reported. We apply Natural GaLore to each optimizer with the rank of 512 and 1024, where the 1B model dimension is 2048.

Pre-training on C4. To evaluate its performance, we apply Natural 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 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 Natural GaLore against LoRA for memory-efficient fine-tuning.

5.1 Comparison with Existing Low-Rank Methods

We first compare Natural 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 Natural GaLore, we set subspace frequency T to 200 and scale factor α to 0.25 across all model sizes in Table **??**. 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 **??**, Natural GaLore outperforms other low-rank methods and achieves comparable performance to full-rank training. We note that for 1B model size, Natural GaLore even outperforms full-rank baseline when $r = 1024$ instead of $r = 512$. Compared to LoRA and ReLoRA, Natural 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 Natural GaLore with Memory-Efficient Optimizers

We demonstrate that Natural GaLore can be applied to various learning algorithms, especially memory-efficient optimizers, to further reduce the memory footprint. We apply Natural 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. **??**, applying Natural GaLore does not significantly affect their convergence. By using Natural 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 Natural

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

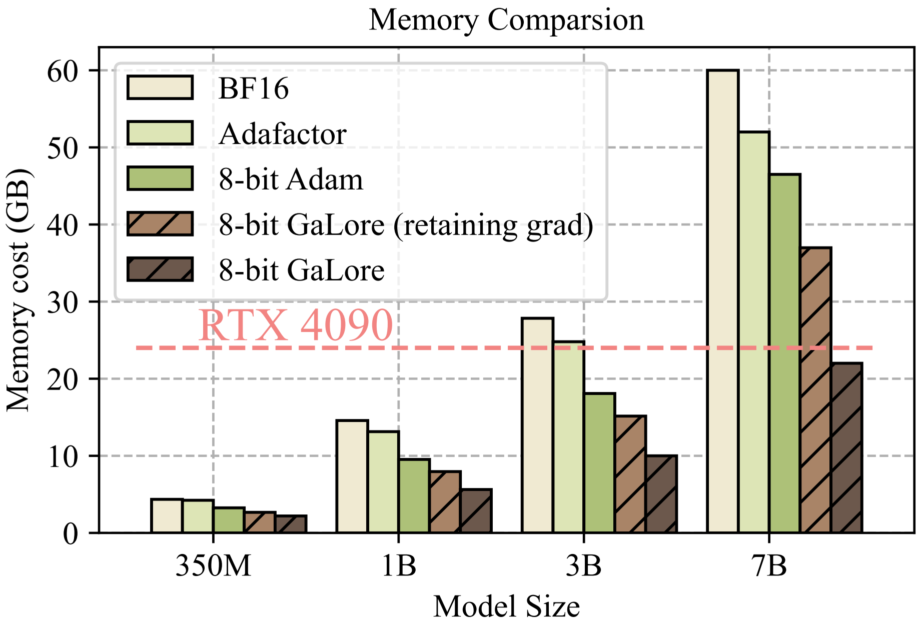

Figure 4: Memory usage for different methods at various model sizes, evaluated with a token batch size of 256. 8-bit Natural GaLore (retaining grad) disables per-layer weight updates but stores weight gradients during training.

Table 4: Evaluating Natural 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
Natural 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
Natural 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

GaLore as Natural 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 Natural GaLore is effective for practical LLM pre-training scenarios. We evaluate Natural 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 Natural GaLore ($r = 1024$) with 8-bit Adam with a single trial without tuning the hyperparameters. As shown in Table **??**, after 150K steps, 8-bit Natural GaLore achieves a perplexity of 14.65, comparable to 8-bit Adam with a perplexity of 14.61.

5.4 Memory-Efficient Fine-Tuning

Natural 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 Natural 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 Natural GaLore. As shown in Table **??**, Natural GaLore achieves better performance than LoRA on most tasks with less memory footprint. This demonstrates that Natural GaLore can serve as a full-stack memory-efficient training strategy for both LLM pre-training and fine-tuning.

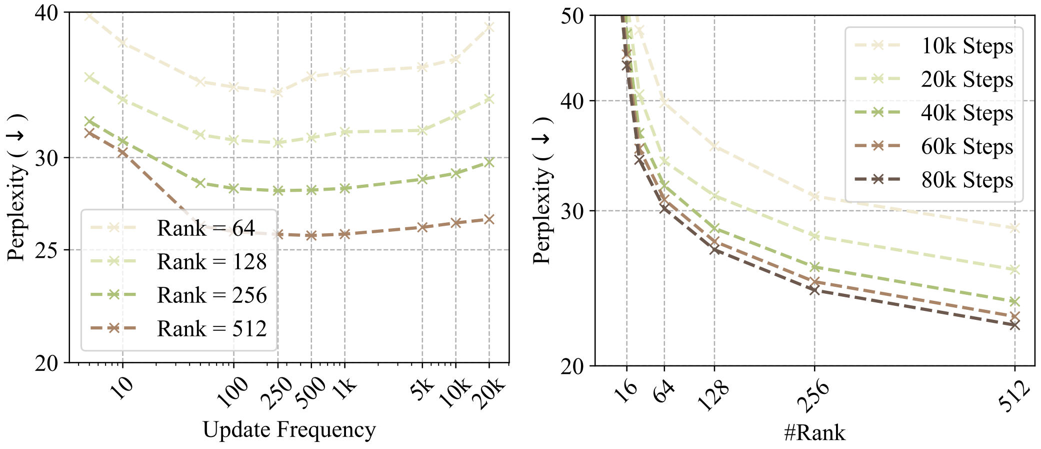

Figure 5: Ablation study of Natural GaLore on 130M models. **Left:** varying subspace update frequency T. **Right:** varying subspace rank and training iterations.

5.5 Measurement of Memory and Throughput

While Table **??** gives the theoretical benefit of Natural 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. **??**, 8-bit Natural 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 Natural GaLore can be used for elastic training **?** 7B models on consumer GPUs such as RTX 4090s.

Specifically, we present the memory breakdown in Fig. **??**. It shows that 8-bit Natural 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 Natural 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 Natural GaLore. We also measure the throughput of the pre-training LLaMA 1B model with 8-bit Natural GaLore and other methods, where the results can be found in the appendix. Particularly, the current implementation of 8-bit Natural GaLore achieves 1019.63 tokens/second, which induces 17% overhead compared to 8-bit Adam implementation. Disabling per-layer weight updates for Natural 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 Natural 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. **??** (left). The reason has been discussed in Sec. **??**. 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. **??** (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 Natural 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 Natural GaLore, a memory-efficient pre-training and fine-tuning strategy for large language models. Natural 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 Natural GaLore, which include (1) applying Natural 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 Natural 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.