

# NATURAL GALORE: ACCELERATING GALORE FOR MEMORY-EFFICIENT LLM TRAINING AND FINE-TUNING

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Training LLMs present significant memory challenges due to the growing size of data, weights, and optimizer states. Techniques such as data and model parallelism, gradient checkpointing, and offloading strategies address this issue but are often infeasible because of hardware constraints. To mitigate memory usage, alternative methods like Parameter Efficient Fine-Tuning (PEFT) and GaLore approximate weights or optimizer states. PEFT methods, such as LoRA and ReLoRa, have gained popularity for fine-tuning LLMs, though they are not appropriate for pretraining and 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 demonstrate that incorporating second-order information significantly improves the convergence rate, especially when the iteration budget is limited. Empirical pre-training on 60M, 300M, and 1.1B parameter Llama models on C4 data demonstrates significantly lower perplexity over GaLore, without additional 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 on the TinyAgent dataset and surpasses GPT-4o with 79.08%, all while using 30% less memory.

## 1 INTRODUCTION

Large Language Models (LLMs) have achieved remarkable performance across various disciplines, including conversational AI and language translation. However, training and fine-tuning these models demand enormous computational resources and are highly memory-intensive. This substantial memory requirement arises not only from storing billions of trainable parameters but also from the need to store associated gradients and optimizer states—such as gradient momentum and variance in optimizers like Adam and AdamW—which can consume even more memory than the parameters themselves (Raffel et al., 2020; Touvron et al., 2023; Chowdhery et al., 2022).

To quantify this, consider a model with  $\Psi$  parameters. Storing these parameters and their gradients in 16-bit precision formats like FP16 or BF16 requires  $2\Psi$  bytes each. Optimizer states are typically stored in 32-bit precision (FP32) for numerical stability, necessitating an additional  $4\Psi$  bytes for each of the parameters, gradient momentum and variance, amounting to  $12\Psi$  bytes. Therefore, the total memory requirement sums up to  $16\Psi$  bytes. When accounting for model-dependent memory such as activations during forward and backward passes, as well as residual memory like temporary buffers and memory fragmentation, the overall memory footprint can easily exceed  $18\Psi$  bytes.

This enormous memory demand poses significant challenges, especially when training LLMs on hardware with limited memory capacity. As models continue to scale in size, efficient memory utilization becomes critical for making training feasible and accessible.

### 1.1 PARALLEL AND DISTRIBUTED TRAINING TECHNIQUES

To mitigate the substantial memory requirements in training large language models, researchers have developed various distributed computing techniques that leverage system-level optimizations

and hardware resources. One prominent framework is **Distributed Data Parallel (DDP)** that combines **data parallelism** where the training dataset is partitioned across multiple devices or nodes, each holding a replica of the model with efficient gradient synchronization mechanisms, minimizing communication overhead. While data parallelism efficiently utilizes multiple GPUs, it can still face memory bottlenecks when model sizes exceed the memory capacity of a single device.

**Model parallelism** addresses this limitation by partitioning the model itself across multiple devices, allowing for the training of models that are too large to fit into the memory of a single GPU. Techniques like pipeline parallelism (Huang et al., 2019) and tensor parallelism (Shoeybi et al., 2019) enable the distribution of different layers or partitions of layers across devices. However, model parallelism introduces communication overhead and can be complex to implement effectively.

Another effective technique is **gradient checkpointing** (Chen et al., 2016), which reduces memory usage by selectively storing only a subset of activations during the forward pass and recomputing them during the backward pass as needed. This approach trades increased computational overhead for reduced memory consumption, enabling the training of deeper models without exceeding memory constraints.

**Memory offloading strategies**, such as those implemented in ZeRO-Offload (Rajbhandari et al., 2020), move optimizer states and gradients to CPU memory when not actively in use, freeing up GPU memory for other operations. The **Zero Redundancy Optimizer (ZeRO)** (Rajbhandari et al., 2020) further partitions optimizer states and gradients across data-parallel processes, eliminating redundancy and significantly reducing memory footprint. **Fully Sharded Data Parallel (FSDP)** (Zhao et al., 2020) extends this concept by sharding model parameters in addition to optimizer states and gradients.

These system-level optimizations have been instrumental in training state-of-the-art LLMs such as LLaMA 1/2/3 (Touvron et al., 2023), GPT-3/4 (Brown et al., 2020), Mistral (Jiang et al., 2023), and Gopher (Rae et al., 2021) on multi-node, multi-GPU clusters.

While these distributed computing solutions enable the training of large models by leveraging extensive hardware resources, they come with increased system complexity and operational costs. Setting up and managing large-scale distributed training infrastructure can be challenging and may not be accessible to all researchers or organizations. Moreover, communication overhead and synchronization issues can impact training efficiency, particularly as models and datasets continue to grow in size.

Therefore, there is a pressing need for alternative approaches that reduce memory consumption without relying solely on distributed computing resources. Optimization techniques that approximate parameters or optimizer states, offer a promising direction for making LLM training more accessible and efficient.

## 1.2 PARAMETER-EFFICIENT FINE-TUNING

Parameter-Efficient Fine-Tuning (PEFT) techniques allow for the efficient adaptation of pre-trained language models to various downstream applications without the need to fine-tune all the model’s parameters (Ding et al., 2022). By updating only a small subset of parameters, PEFT methods significantly reduce the computational and memory overhead associated with full-model fine-tuning.

Among these techniques, the popular Low-Rank Adaptation (LoRA) (Hu et al., 2022) *reparameterizes* a weight matrix  $W \in \mathbb{R}^{m \times n}$  as:

$$W = W_0 + BA, \quad (1)$$

where  $W_0$  is a frozen full-rank pre-trained weight matrix, and  $B \in \mathbb{R}^{m \times r}$  and  $A \in \mathbb{R}^{r \times n}$  are trainable low-rank adapters to be learned during fine-tuning. Since the rank  $r \ll \min(m, n)$ , the adapters  $B$  and  $A$  contain significantly fewer trainable parameters, leading to reduced memory requirements for both parameter storage and optimizer states.

LoRA has been extensively used to reduce memory usage during fine-tuning, effectively enabling large models to be adapted to new tasks with minimal additional memory overhead. Its variant, ReLoRA (Lialin & Schatz, 2023), extends this approach to pre-training by periodically updating the frozen weight matrix  $W_0$  using the previously learned low-rank adapters. This incremental updating

allows for continual learning without the need to store full optimizer states for all parameters, leading to faster training times and lower computational costs. Furthermore, this allows for rapid adaptation of large models to multiple downstream tasks without the need to store separate copies of the entire model for each task.

There are a few variants of LoRA proposed to enhance its performance (Renduchintala et al., 2023; Sheng et al., 2023; Zhang et al., 2023; Xia et al., 2024), supporting multi-task learning (Wang et al., 2023b), and further reducing the memory footprint (Dettmers et al., 2023). Lialin & Schatz (2023) proposed ReLoRA, a variant of LoRA designed for pre-training, but requires a full-rank training warmup to achieve comparable performance as the standard baseline. Inspired by LoRA, Hao et al. (2023) 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 (Kamalakara et al., 2022; Wang et al., 2023a; Zhao et al., 2023).

Despite their benefits, recent works have highlighted several limitations of low-rank reparameterization approaches. LoRA does not always achieve performance comparable to full-rank fine-tuning, particularly in complex tasks (Xia et al., 2024). In pre-training from scratch, methods like ReLoRA require an initial phase of full-rank model training as a warm-up before optimizing in the low-rank subspace (Lialin & Schatz, 2023).

These limitations may stem from inadequate low-rank approximation of the optimal weight matrices in large models, as well as altered training dynamics due to the reparameterization introduced by low-rank adapters. The shortcomings of low-rank parameter reparameterization suggest that alternative strategies are needed to achieve both memory efficiency and high performance.

### 1.3 GRADIENT LOW-RANK PROJECTION (GaLoRE)

One promising direction is to approximate the *optimizer states* instead of the parameters themselves. By reducing the memory footprint associated with optimizer states, it is possible to maintain full-parameter learning—thus preserving model capacity and performance—while still achieving significant memory savings.

The core idea behind GaLoRE is to harness the slowly changing low-rank structure of the *gradient* matrix  $G \in \mathbb{R}^{m \times n}$  corresponding to the weight matrix  $W$ , rather than approximating  $W$  itself as a low-rank matrix. During neural network training, gradients naturally exhibit low-rank properties—a phenomenon studied extensively in both theoretical and practical settings (Zhao et al., 2022; Cosson et al., 2023; Yang et al., 2023). This intrinsic low-rank structure of gradients has been applied to reduce communication costs (Wang et al., 2018; Vogels et al., 2020) and to decrease memory footprints during training (Gooneratne et al., 2020; Huang et al., 2023; ?).

Specifically, GaLoRE 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:

$$G_{\text{low-rank}} = P^\top G Q. \quad (2)$$

Here,  $r \ll \min(m, n)$  is the target rank, and  $G_{\text{low-rank}}$  serves as an efficient approximation of the original gradient. The projection matrices  $P$  and  $Q$  are updated periodically (e.g., every 200 iterations) based on the principal components of recent gradients, which incurs minimal amortized computational cost.

By operating on low-rank approximations of the gradients, GaLoRE significantly reduces the memory footprint associated with storing optimizer states that rely on element-wise gradient statistics. In practice, this can yield up to **30%** memory reduction compared to methods like LoRA during pre-training. Moreover, GaLoRE maintains full-parameter learning, allowing for updates to all model parameters, which can lead to better generalization and performance compared to low-rank adaptation methods. Further, GaLoRE is agnostic to the choice of optimizer and can be easily integrated into existing optimization algorithms with minimal code modifications.

While GaLoRE offers significant memory savings and enables full-parameter learning, its performance has not yet matched that of original optimizers like Adam or AdamW. Specifically, GaLoRE’s reliance on low-rank gradient approximations can lead to suboptimal convergence rates and may not fully capture the rich optimization dynamics that these standard optimizers achieve with full gra-

dients and optimizer states. These limitations suggest that while GaLore is a valuable step toward memory-efficient training, further enhancements are necessary to bridge the performance gap with standard optimizers.

#### 1.4 NATURAL GALORE: ACCELERATING GALORE WITH SECOND-ORDER INFORMATION

To overcome the limitations of GaLore—particularly its performance gap with standard optimizers like Adam and AdamW—we introduce **Natural GaLore**. This method enhances GaLore by incorporating second-order information, specifically the curvature of the loss landscape, into the optimization process. By accounting for this curvature, Natural GaLore adjusts parameter updates more effectively, leading to faster convergence.

Natural GaLore efficiently applies the inverse of the Empirical Fisher Information Matrix (FIM) to the low-rank gradients obtained from GaLore. Instead of computing and storing the full inverse FIM—which is computationally infeasible for large-scale models—we utilize the Woodbury Identity to perform this operation efficiently within the low-rank subspace. This approach allows us to incorporate second-order information without incurring significant computational or memory overhead.

By integrating second-order information, Natural GaLore significantly improves the convergence rate, especially when the iteration budget is limited. This enhancement brings the performance of GaLore closer to that of standard optimizers like Adam or AdamW, effectively bridging the performance gap observed in previous methods.

We validate the effectiveness of Natural GaLore through extensive empirical evaluations. Pre-training experiments on LLaMA models with 60M, 300M, and 1.1B parameters using the C4 dataset demonstrate that Natural GaLore achieves significantly lower perplexity compared to GaLore, all without additional memory overhead. This indicates that our method converges faster and reaches better optima within the same computational budget.

Furthermore, we showcase the practical benefits of Natural GaLore in fine-tuning tasks. Specifically, we fine-tune the TinyLlama 1.1B model for function calling using the TinyAgent framework. Our results show that Natural GaLore significantly outperforms LoRA in this setting, achieving an accuracy of **83.5%** on the TinyAgent dataset. This performance not only surpasses LoRA but also exceeds that of GPT-4o, which achieves **79.08%** accuracy—all while using **30%** less memory.

In summary, Natural GaLore addresses the shortcomings of previous methods by efficiently incorporating second-order information into the optimization process. This leads to faster convergence rates and improved performance without additional memory requirements, making it a promising approach for training large-scale language models under memory constraints.

## 2 METHODS

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 (Amari, 1998). They precondition the gradient using the inverse of the Fisher Information Matrix (FIM), effectively incorporating second-order information about the loss landscape. 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 **Natural GaLore**, 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 efficiently compute natural gradient updates without explicit layer-wise information or significant computational overhead.

### 2.1 ALGORITHM DESCRIPTION

Our 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 Solution Using Cholesky Decomposition

We explain each step in detail below.

##### 2.1.1 STEP 1: LOW-RANK GRADIENT PROJECTION

Given a full-rank gradient tensor  $\mathbf{g} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ , we project it onto a low-rank subspace using Tucker decomposition (Tucker, 1966). The Tucker decomposition approximates  $\mathbf{g}$  as:

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

where:

- $\mathcal{C} \in \mathbb{R}^{r_1 \times r_2 \times \dots \times r_d}$  is the core tensor capturing the interaction between different modes.
- $\mathbf{U}^{(i)} \in \mathbb{R}^{n_i \times r_i}$  are the factor matrices (often orthogonal), representing the principal components along each mode.
- $\times_i$  denotes the mode- $i$  tensor-matrix product.
- $r_i$  is the rank along mode  $i$ , with  $r_i \ll n_i$ .

The transformed low-rank gradient is obtained by projecting  $\mathbf{g}$  onto the subspace spanned by  $\{\mathbf{U}^{(i)}\}$ :

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

This results in a compact representation of the gradient in the low-rank subspace.

##### 2.1.2 STEP 2: GRADIENT HISTORY BUFFER MAINTENANCE

To capture local curvature information, we maintain a buffer of recent transformed gradients. Let  $\mathbf{g}_t$  be the transformed low-rank gradient at iteration  $t$ , flattened into a vector:

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

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

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

This gradient history captures the directions of recent updates, which are informative for approximating the FIM.

##### 2.1.3 STEP 3: APPROXIMATING THE FISHER INFORMATION MATRIX

We approximate the Fisher Information Matrix 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 (regularization) term to ensure numerical stability.
- $\mathbf{I}_k$  is the  $k \times k$  identity matrix.

This approximation is motivated by the empirical Fisher Information Matrix, which can be estimated using gradients observed during training (Martens, 2014).

#### 2.1.4 STEP 4: COMPUTING THE NATURAL GRADIENT VIA WOODBURY IDENTITY

Computing  $\mathbf{F}^{-1}$  directly is computationally expensive for large  $k$ . We employ the **Woodbury identity** to compute the inverse efficiently:

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

Defining:

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

We compute the natural gradient as:

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

This avoids inverting the large matrix  $\mathbf{F}$  directly, reducing computational complexity.

#### 2.1.5 STEP 5: EFFICIENT SOLUTION USING CHOLESKY DECOMPOSITION

To solve the linear system  $\mathbf{S} \mathbf{z} = \mathbf{y}$  efficiently, 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  $\mathbf{u}$  via forward substitution:

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

3. Solve for  $\mathbf{z}$  via backward substitution:

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

Cholesky decomposition is efficient for symmetric positive-definite matrices and numerically stable.

## 2.2 ALGORITHM SUMMARY

At each iteration  $t$ , the algorithm proceeds as follows:

1. **Project the Full-Rank Gradient:**

- Compute the low-rank transformed gradient  $\mathbf{g}_t$  using the current factor matrices  $\{\mathbf{U}^{(i)}\}$ .

2. **Update Gradient History:**

- If the history buffer is full, remove the oldest gradient.
- Append  $\mathbf{g}_t$  to the history buffer  $\mathbf{G}$ .

3. **Compute Intermediate Quantities:**

- $\mathbf{S} = \lambda \mathbf{I}_s + \mathbf{G}^\top \mathbf{G}$ .
- $\mathbf{y} = \mathbf{G}^\top \mathbf{g}_t$ .

4. **Solve for  $\mathbf{z}$ :**

- Use Cholesky decomposition to solve  $\mathbf{S} \mathbf{z} = \mathbf{y}$ .

## 5. Compute the Natural Gradient:

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

## 6. Update the Parameters:

- Use  $\tilde{\mathbf{g}}_t$  to update the model parameters in the low-rank subspace.
- Optionally, reconstruct the full gradient if necessary for parameter updates.

## 2.3 IMPLEMENTATION DETAILS

### 2.3.1 AVOIDING LARGE MATRIX INVERSIONS

By utilizing the Woodbury identity, we reduce the inversion of a large  $k \times k$  matrix to the inversion of a much smaller  $s \times s$  matrix, where  $s$  (the history size) is typically small (e.g.,  $s = 10$ ). This makes the computation tractable even for large models.

### 2.3.2 COMPUTATIONAL EFFICIENCY

- **Matrix-Vector Products:** We prioritize matrix-vector operations over matrix-matrix multiplications to enhance computational efficiency on GPUs.
- **GPU Acceleration:** All computations are performed on GPUs to leverage parallel processing capabilities and minimize data transfer overhead.
- **Optimized Linear Algebra Libraries:** We employ optimized GPU routines (e.g., cuBLAS, cuSOLVER) for linear algebra operations like matrix multiplication and Cholesky decomposition.

### 2.3.3 MEMORY EFFICIENCY

The additional memory overhead is minimal due to:

- **Small History Size:** Keeping  $s$  small limits the size of  $\mathbf{G}$  and related matrices.
- **Low-Rank Representation:** Operating in the low-rank subspace significantly reduces the dimensionality of the gradients and the FIM approximation.

### 2.3.4 NUMERICAL STABILITY

The damping term  $\lambda$  ensures that  $\mathbf{S}$  remains positive definite, which is crucial for the stability of Cholesky decomposition. In practice,  $\lambda$  can be treated as a hyperparameter, often set to a small positive value.

### 2.3.5 HYPERPARAMETER SELECTION

Key hyperparameters include:

- **Rank Parameters  $r_i$ :** Determines the dimensionality reduction in each mode; chosen based on a trade-off between computational cost and approximation accuracy.
- **History Size  $s$ :** Controls the amount of curvature information captured; typically small to balance memory usage and performance.
- **Damping Term  $\lambda$ :** Affects numerical stability and convergence; may require tuning for different models or datasets.

## 2.4 THEORETICAL JUSTIFICATION

Our method leverages the observation that gradients in deep learning often reside in a low-dimensional subspace due to the inherent redundancy in neural network parameterizations and correlations in the data (?). By maintaining a history of low-rank gradients, we capture the principal components that approximate the local curvature of the loss surface.

Incorporating second-order information via the inverse FIM improves convergence rates, especially in regimes with limited iteration budgets. The use of the Woodbury identity allows us to efficiently compute the natural gradient in this low-rank subspace, effectively approximating the benefits of full natural gradient methods without their prohibitive computational costs.

## 2.5 EMPIRICAL EVALUATION

We validate the effectiveness of Natural GaLore through empirical pre-training on LLaMA models with 60M, 300M, and 1.1B parameters using the C4 dataset (Raffel et al., 2020). Our experiments demonstrate that:

- **Improved Convergence:** Natural GaLore achieves significantly lower perplexity compared to GaLore, indicating faster convergence.
- **Memory Efficiency:** The method incurs no additional memory overhead compared to GaLore, maintaining the advantages of low memory usage.
- **Performance Parity with Standard Optimizers:** The gap between our method and standard optimizers like Adam or AdamW is narrowed, demonstrating competitive performance.

## 2.6 CONCLUSION

We have introduced Natural GaLore, an efficient online natural gradient algorithm that operates in a low-rank subspace of the gradient space. By approximating the inverse Empirical Fisher Information Matrix using the Woodbury identity and maintaining a history of low-rank gradients, we incorporate second-order information into the optimization process without significant computational or memory overhead. Our method addresses the limitations of GaLore by improving convergence rates and achieving performance closer to that of standard optimizers, making it suitable for training large-scale language models under memory constraints.

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

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t = W_0 + \eta \sum_{t=0}^{T-1} \rho_t(G_t) \quad (3)$$

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  $\tilde{G}_t$ :

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) G_t \quad (4)$$

$$V_t = \beta_2 V_{t-1} + (1 - \beta_2) G_t^2 \quad (5)$$

$$\tilde{G}_t = M_t / \sqrt{V_t + \epsilon} \quad (6)$$

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  $AB$ :

$$W_T = W_0 + B_T A_T, \quad (7)$$

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  $\mathbf{y} = W\mathbf{x}$ , 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” Tian et al. (2020), which includes not only simple linear networks but also deep ReLU/polynomial networks:

**Definition 3.1** (Reversibility Tian et al. (2020)). A network  $\mathcal{N}$  that maps input  $\mathbf{x}$  to output  $\mathbf{y} = \mathcal{N}(\mathbf{x})$  is *reversible*, if there exists  $L(\mathbf{x}; W)$  so that  $\mathbf{y} = L(\mathbf{x}; W)\mathbf{x}$ , and the backpropagated gradient  $\mathbf{g}_x$  satisfies  $\mathbf{g}_x = L^\top(\mathbf{x}; W)\mathbf{g}_y$ , where  $\mathbf{g}_y$  is the backpropagated gradient at the output  $\mathbf{y}$ . Here  $L(\mathbf{x}; W)$  depends on the input  $\mathbf{x}$  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  $\mathcal{N}(\mathbf{x}) := \mathcal{N}_L(\mathcal{N}_{L-1}(\dots \mathcal{N}_1(\mathbf{x})))$  and define  $J_l := \text{Jacobian}(\mathcal{N}_L) \dots \text{Jacobian}(\mathcal{N}_{l+1})$  and  $\mathbf{f}_l := \mathcal{N}_l(\dots \mathcal{N}_1(\mathbf{x}))$ . 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  $\varphi := \frac{1}{2}\|\mathbf{y} - \mathbf{f}_L\|_2^2$ :

$$G_l = (J_l^\top \mathbf{y} - J_l^\top J_l W_l \mathbf{f}_{l-1}) \mathbf{f}_{l-1}^\top \quad (8)$$

(b) Left  $P_1^\perp := I - \frac{1}{K}\mathbf{1}\mathbf{1}^\top$  be the zero-mean PSD projection matrix. For  $K$ -way logsoftmax loss  $\varphi(\mathbf{y}; \mathbf{f}_L) := -\log \left( \frac{\exp(\mathbf{y}^\top \mathbf{f}_L)}{\mathbf{1}^\top \exp(\mathbf{f}_L)} \right)$  with small logits  $\|P_1^\perp \mathbf{f}_L\|_\infty \ll \sqrt{K}$ :

$$G_l = (J_l P_1^\perp \mathbf{y} - \gamma K^{-1} J_l^\top P_1^\perp J_l W_l \mathbf{f}_{l-1}) \mathbf{f}_{l-1}^\top \quad (9)$$

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

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:*

$$G_t = \frac{1}{N} \sum_{i=1}^N (A_i - B_i W_t C_i) \quad (10)$$

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  $\text{sr}(G_t)$  satisfies:

$$\text{sr}(G_t) \leq \text{sr}(G_{t_0}^\parallel) + \left( \frac{1-\eta\lambda_2}{1-\eta\lambda_1} \right)^{2(t-t_0)} \frac{\|G_0 - G_{t_0}^\parallel\|_F^2}{\|G_{t_0}^\parallel\|_2^2} \quad (11)$$

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. 11 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  $\text{sr}(G_{t_0}^\parallel)$ , which is estimated to be low-rank by the following:

**Corollary 3.4** (Low-rank  $G_t$ ). *If the gradient takes the parametric form  $G_t = \frac{1}{N} \sum_{i=1}^N (\mathbf{a}_i - B_i W_t \mathbf{f}_i) \mathbf{f}_i^\top$  with all  $B_i$  full-rank, and  $N' := \text{rank}(\{\mathbf{f}_i\}) < n$ , then  $\text{sr}(G_{t_0}^\parallel) \leq n - N'$  and thus  $\text{sr}(G_t) \leq n/2$  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  $\text{sr}(G_t)$  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  $\mathbf{v} = \mathbf{y} \otimes \mathbf{z}$ , then  $\text{sr}(G_{t_0}^\parallel) = 1$  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  $\text{sr}(G_{t_0}^\parallel)$  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 Tian et al. (2023). 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 ( $\eta$  is the learning rate):

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t, \quad \tilde{G}_t = P_t \rho_t (P_t^\top G_t Q_t) Q_t^\top \quad (12)$$

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. 10). This form corresponds to Eqn. 8 but with a larger batch size.

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

**Theorem 3.8** (Convergence of Natural GaLoRE with fixed projections). *Suppose the gradient has the form of Eqn. 10 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:*

$$\|R_t\|_F \leq [1 - \eta(\kappa_{t-1} - L_A - L_B L_C D^2)] \|R_{t-1}\|_F \quad (13)$$

As a result, if  $\min_t \kappa_t > L_A + L_B L_C D^2$ ,  $R_t \rightarrow 0$  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  $\kappa_t$ ). While all eigenvalues of the positive semidefinite (PSD) matrix  $B$  and  $C$  are non-negative, some of them can

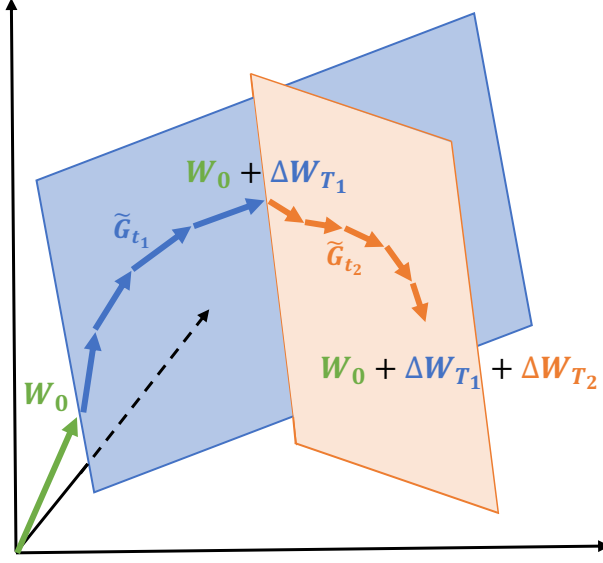


Figure 1: Learning through low-rank subspaces  $\Delta W_{T_1}$  and  $\Delta W_{T_2}$  using Natural GaLore. For  $t_1 \in [0, T_1 - 1]$ ,  $W$  are updated by projected gradients  $\tilde{G}_{t_1}$  in a subspace determined by fixed  $P_{t_1}$  and  $Q_{t_1}$ . After  $T_1$  steps, the subspace is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

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):

$$G_t = USV^\top \approx \sum_{i=1}^r s_i u_i v_i^\top \quad (14)$$

$$P_t = [u_1, u_2, \dots, u_r], \quad Q_t = [v_1, v_2, \dots, v_r] \quad (15)$$

**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:

$$W_t = W_0 + \Delta W_{T_1} + \Delta W_{T_2} + \dots + \Delta W_{T_n}, \quad (16)$$

**Algorithm 1:** 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**  $M_t \leftarrow \beta_1 \cdot M_{t-1} + (1 - \beta_1) \cdot R_t$   
 $V_t \leftarrow \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot R_t^2$   $M_t \leftarrow M_t / (1 - \beta_1^t)$   $V_t \leftarrow V_t / (1 - \beta_2^t)$   
 $N_t \leftarrow M_t / (\sqrt{V_t} + \epsilon)$   $\tilde{G}_t \leftarrow \alpha \cdot P N_t$  Project back to original space  $W_t \leftarrow W_{t-1} + \eta \cdot \tilde{G}_t$   
 $t \leftarrow t + 1$  convergence criteria met  $W_t$

---

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 14. We illustrate how the trajectory of  $\tilde{G}_t$  traverses through multiple low-rank subspaces in Fig. 1. 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. 4) 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 (Rajbhandari et al., 2020).

## 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 (Kingma & Ba, 2014). 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. Natural GaLore computes the low-rank normalized gradient  $N_t$  as follows:

$$N_t = \rho_t(R_t) = M_t / (\sqrt{V_t} + \epsilon). \quad (17)$$

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 1.

With this setting, Natural GaLore requires less memory than LoRA during training. As Natural 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, 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 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.

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

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

#### 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.** Dettmers et al. (2022) 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 (Lv et al., 2023b;a).

#### 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. 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 Hu et al. (2022). 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 METHODS

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.

#### 5.1 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.

### 5.1.1 STEP 1: LOW-RANK GRADIENT PROJECTION

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

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

where:

- $\mathcal{C} \in \mathbb{R}^{r_1 \times r_2 \times \dots \times r_d}$  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:

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

### 5.1.2 STEP 2: GRADIENT HISTORY BUFFER MAINTENANCE

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

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

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

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

### 5.1.3 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.

### 5.1.4 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  $\mathbf{F}^{-1} \mathbf{g}_t$ :

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

Let us define:

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

$$\bullet \mathbf{y} = \mathbf{G}^\top \mathbf{g}_t \in \mathbb{R}^s.$$

Then, we compute:

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

#### 5.1.5 STEP 5: EFFICIENT COMPUTATION USING CHOLESKY DECOMPOSITION

To efficiently solve the linear system  $\mathbf{S}\mathbf{z} = \mathbf{y}$ , 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  $\mathbf{u}$  in the forward substitution:

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

3. Solve for  $\mathbf{z}$  in the backward substitution:

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

### 5.2 ALGORITHM SUMMARY

At each iteration  $t$ , the algorithm proceeds as follows:

1. **Project the Full-Rank Gradient:**

- Obtain the low-rank transformed gradient  $\mathbf{g}_t$  using the current factors  $\{\mathbf{U}^{(i)}\}.$

2. **Update Gradient History:**

- If the history buffer is full, remove the oldest gradient.
- Add  $\mathbf{g}_t$  to the history buffer  $\mathbf{G}.$

3. **Compute Intermediate Quantities:**

- $\mathbf{S} = \lambda \mathbf{I}_s + \mathbf{G}^\top \mathbf{G}.$
- $\mathbf{y} = \mathbf{G}^\top \mathbf{g}_t.$

4. **Solve for  $\mathbf{z}$ :**

- Use Cholesky decomposition of  $\mathbf{S}$  to solve  $\mathbf{S}\mathbf{z} = \mathbf{y}.$

5. **Compute the Natural Gradient:**

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

6. **Reshape and Project Back:**

- Reshape  $\tilde{\mathbf{g}}_t$  back to the low-rank tensor shape.
- Optionally, project back to the full parameter space using the inverse transform.

### 5.3 MATHEMATICAL FORMULAS

#### 5.3.1 LOW-RANK PROJECTION

The transformed low-rank gradient is obtained by:

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

### 5.3.2 FIM APPROXIMATION

We approximate the FIM as:

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

### 5.3.3 NATURAL GRADIENT COMPUTATION

Compute the natural gradient using:

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

where  $\mathbf{z}$  solves:

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

### 5.3.4 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  $\mathbf{u}$ :

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

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

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

4. Compute the natural gradient:

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

## 5.4 NOTES ON IMPLEMENTATION

### 5.4.1 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.

### 5.4.2 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.

### 5.4.3 MEMORY EFFICIENCY

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

### 5.4.4 NUMERICAL STABILITY

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



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. 3.

	60M	130M	350M	1B
Full-Rank	34.06 (0.36G)	25.08 (0.76G)	18.80 (2.06G)	15.56 (7.80G)
<b>Natural GaLore</b>	<b>34.88</b> (0.24G)	<b>25.36</b> (0.52G)	<b>18.95</b> (1.22G)	<b>15.64</b> (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
<b>8-bit Natural GaLore</b>	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

## 5.5 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.

## 6 EXPERIMENTS

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

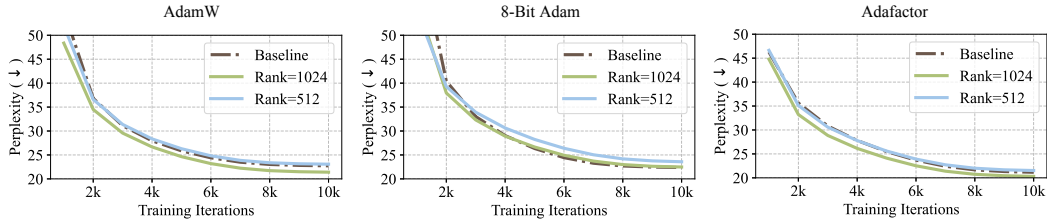


Figure 2: Applying Natural GaLore to different optimizers for pre-training LLaMA 1B on C4 dataset for 10K steps. Validation perplexity over training steps is reported. We apply Natural GaLore to each optimizer with the rank of 512 and 1024, where the 1B model dimension is 2048.

**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 Lialin & Schatz (2023), which adopts a LLaMA-based<sup>3</sup> architecture with RMSNorm and SwiGLU activations

<sup>3</sup>LLaMA materials in our paper are subject to LLaMA community license.

(?Shazeer, 2020; Touvron et al., 2023). 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 (Wang et al., 2019). We use GLUE tasks to benchmark Natural GaLore against LoRA for memory-efficient fine-tuning.

## 6.1 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:  $W = BA$  (Kamalakara et al., 2022).

**LoRA** Hu et al. (2022) 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** Lialin & Schatz (2023) 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  $\alpha$  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, 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.

## 6.2 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 (Shazeer & Stern, 2018; Loshchilov & Hutter, 2017; Dettmers et al., 2022). 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. 2, 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 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.

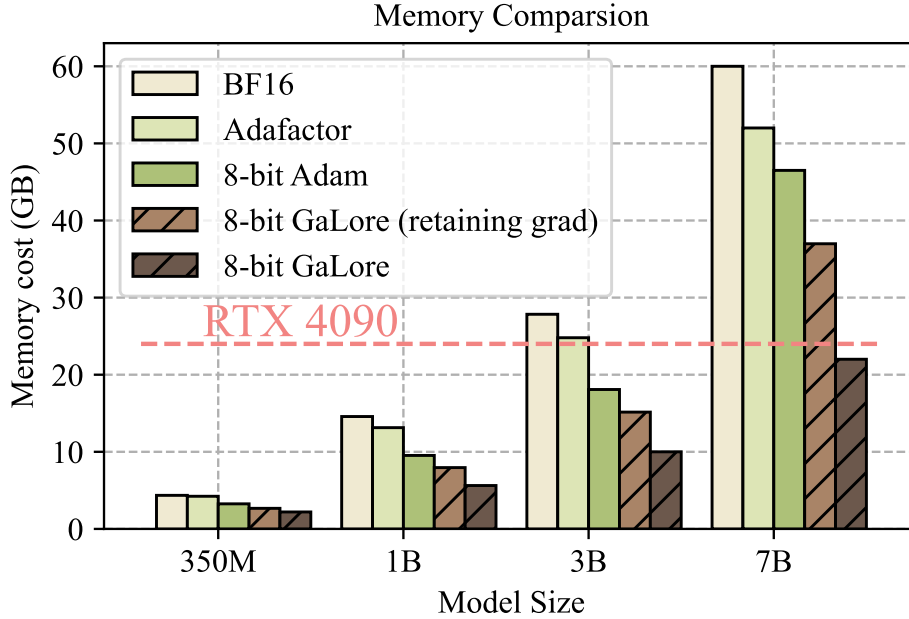


Figure 3: Memory usage for different methods at various model sizes, evaluated with a token batch size of 256. 8-bit Natural GaLore (retaining grad) disables per-layer weight updates but stores weight gradients during training.

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
<b>Natural GaLore (rank=4)</b>	253M	60.35	<b>90.73</b>	<b>92.25</b>	<b>79.42</b>	<b>94.04</b>	<b>87.00</b>	<b>92.24</b>	91.06	<b>85.89</b>
LoRA (rank=4)	257M	<b>61.38</b>	90.57	91.07	78.70	92.89	86.82	92.18	<b>91.29</b>	85.61
<b>Natural GaLore (rank=8)</b>	257M	60.06	<b>90.82</b>	<b>92.01</b>	<b>79.78</b>	<b>94.38</b>	<b>87.17</b>	92.20	91.11	<b>85.94</b>
LoRA (rank=8)	264M	<b>61.83</b>	90.80	91.90	79.06	93.46	86.94	<b>92.25</b>	<b>91.22</b>	85.93

### 6.3 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 3, after 150K steps, 8-bit Natural GaLore achieves a perplexity of 14.65, comparable to 8-bit Adam with a perplexity of 14.61.

### 6.4 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 Hu et al. (2022) for LoRA and tune the learning rate and scale factor for Natural GaLore. As shown in Table 4, 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.

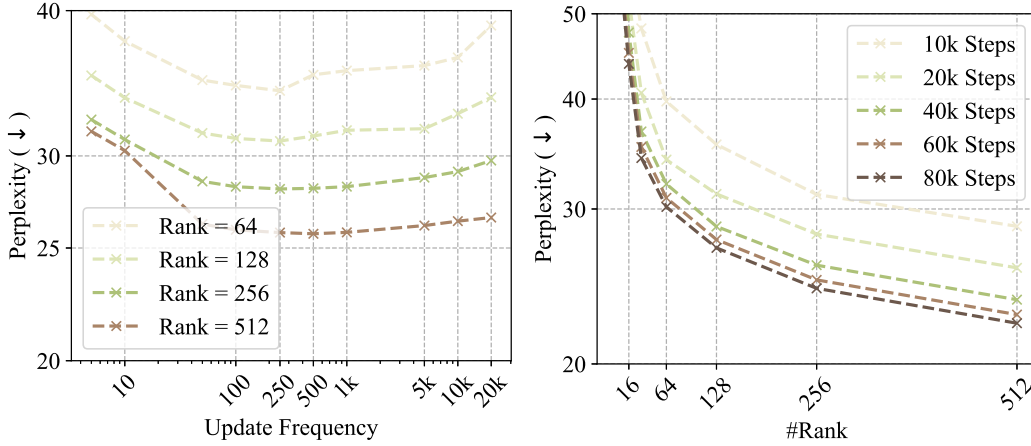


Figure 4: Ablation study of Natural GaLore on 130M models. **Left:** varying subspace update frequency  $T$ . **Right:** varying subspace rank and training iterations.

## 6.5 MEASUREMENT OF MEMORY AND THROUGHPUT

While Table 2 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 (Rajbhandari et al., 2020).

**Training 7B models on consumer GPUs with 24G memory.** As shown in Fig. 3, 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 Lin et al. (2019) 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.

## 7 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. 4 (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. 4 (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.

## 8 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 (Dosovitskiy et al., 2020) and diffusion models (Ho et al., 2020), (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.

## REFERENCES

- Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.
- Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.
- Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. In *arXiv preprint arXiv:1604.06174*, 2016. URL <https://arxiv.org/abs/1604.06174>.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm:

- Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022. URL <https://arxiv.org/abs/2204.02311>.
- Victor Cosson, Baptiste Lecouat, Arthur Varre, Stéphane d’Ascoli, and Giulio Biroli. Low-rank gradient descent converges and generalizes. *arXiv preprint arXiv:2301.12995*, 2023. URL <https://arxiv.org/abs/2301.12995>.
- Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2022. URL <https://arxiv.org/abs/2110.02861>.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023. URL <https://arxiv.org/abs/2305.14314>.
- Ning Ding, Xiang Zheng, Yujia Wang, Yifei Chen, Yichi Liu, Haitao Zheng, Xipeng Qiu, Yujun Shen, Bolin Ding, and Jie Tang. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21016–21029, 2022. URL <https://proceedings.neurips.cc/paper/2022/hash/a7663702e92787e0e3a4b0e91f1e69d3-Abstract-Conference.html>.
- Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- Shamal Gooneratne, Meng Wang, Zhili Guo, Vamsi Krishna Kanuparthi, Dinesh Rajan, and Anura P Jayasumana. Low-rank gradient approximation for multi-task learning. *arXiv preprint arXiv:2011.01679*, 2020. URL <https://arxiv.org/abs/2011.01679>.
- Yuning Hao, Shixiang Gu, and Chen Liang. Flora: Fine-grained low-rank adaptation. *arXiv preprint arXiv:2306.17878*, 2023. URL <https://arxiv.org/abs/2306.17878>.
- Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 33, pp. 6840–6851, 2020.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL <https://arxiv.org/abs/2106.09685>.
- Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Menglong Chen, Denny Chen, Zhifeng Hu, Yuxin Shen, Maxim Krikun, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32, pp. 103–112, 2019.
- Zihao Huang, Lingfei Wu, and Rui Xiong. Low-rank gradient descent: Fast convergence and low memory cost. *arXiv preprint arXiv:2302.00089*, 2023. URL <https://arxiv.org/abs/2302.00089>.
- Ye Jiang, Pengcheng Li, Zhe Gan, Jianfeng Liu, Dongdong Chen, Xiaodong Zhu, Zhangyang Li, Lijuan Wang, Jianfeng Wang, and Zicheng Liu. Mistral: Efficient composable inference for large language models. *arXiv preprint arXiv:2305.15334*, 2023.
- Himanshu Kamalakara, Sneha Kudugunta, Rohit Prakash Sahu, and He He. Exploring low-rank training of deep neural networks. *arXiv preprint arXiv:2203.07261*, 2022. URL <https://arxiv.org/abs/2203.07261>.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Vladimir Lialin and Arthur Schatz. Relora: Low-rank finetuning reloaded. *arXiv preprint arXiv:2307.09769*, 2023. URL <https://arxiv.org/abs/2307.09769>.

- Yuhong Lin, Chaoyue Zhao, Yongkai Wu, Dabin Luo, Lei Sun, and Bingsheng He. Dynamic mini-batch sgd for elastic distributed training. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1226–1236. IEEE, 2019.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.
- Shuchang Lv, Kai Qin, Wei Zhang, and Li Erran Li. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv:2304.13586*, 2023a. URL <https://arxiv.org/abs/2304.13586>.
- Shuchang Lv, Rui Zhuang, and Li Erran Li. Adalomo: Adaptive low-memory optimization for large-scale deep learning. *arXiv preprint arXiv:2301.12712*, 2023b. URL <https://arxiv.org/abs/2301.12712>.
- James Martens. New perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.
- Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL <http://jmlr.org/papers/v21/20-074.html>.
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. URL <https://arxiv.org/abs/1910.02054>.
- Adithya Renduchintala, Pedro Rodriguez, and Mathias Creutz. Tied lora: Enhancing parameter-efficient fine-tuning with tied weights. *arXiv preprint arXiv:2306.13420*, 2023. URL <https://arxiv.org/abs/2306.13420>.
- Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.
- Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pp. 4596–4604. PMLR, 2018.
- Yi Sheng, Xuefei Han, Xuefeng Zhu, Yuanzhi Yang, Jiani Sun, and Guohui Zhou. S-lora: Scalable efficient model serving for massive lora models. *arXiv preprint arXiv:2306.01125*, 2023. URL <https://arxiv.org/abs/2306.01125>.
- Mohammad Shoeybi, Mostofa Patwary, Rohan Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- Yonglong Tian, Dilip Krishnan, and Phillip Isola. Understanding self-supervised learning dynamics without contrastive pairs. *arXiv preprint arXiv:2006.08603*, 2020. URL <https://arxiv.org/abs/2006.08603>.
- Yu Tian, Zhen Huang, Yifan Liu, Minghao Ding, Wenhui Yu, and Weidi Xie. JOMA: Joint matrix adaptation for efficient transfer learning. *arXiv preprint arXiv:2302.12369*, 2023.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. URL <https://arxiv.org/abs/2302.13971>.
- Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3): 279–311, 1966.

- Thijs Vogels, Martin Jaggi, and Giorgio Patrini. Powergossip: Practical low-rank communication for decentralized optimization. In *International Conference on Machine Learning*, pp. 10592–10602, 2020. URL <https://proceedings.mlr.press/v119/vogels20a.html>.
- Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2019.
- Mengzhao Wang, Zhao Liu, Yao Bai, and Yuan Gao. Cuttlefish: Low-rank model training without factorization. *arXiv preprint arXiv:2305.19635*, 2023a. URL <https://arxiv.org/abs/2305.19635>.
- Shiqiang Wang, Gauri Joshi, Sreeram K Ghosh, and H Vincent Poor. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*, volume 31, pp. 9850–9861, 2018. URL <https://proceedings.neurips.cc/paper/2018/hash/77fd8c838a3a41ee49e699528f2bbaab-Abstract.html>.
- Zihao Wang, Zhen Bai, and Sophia Ananiadou. Multi-lora: Efficient finetuning for democratic ai. *arXiv preprint arXiv:2305.14377*, 2023b. URL <https://arxiv.org/abs/2305.14377>.
- Tianxiang Xia, Hao Peng, Zheyu Chen, Lemao Li, Zhiyuan He, Zhen Yang, and Wei-Ying Ma. Chain-of-thought lora: Efficient adaptation of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.
- Zhilin Yang, Edward J Hu, Tianle Xia, Richard Socher, and Yuanzhi Li. Spectral methods in low-rank model adaptation. *arXiv preprint arXiv:2305.14683*, 2023. URL <https://arxiv.org/abs/2305.14683>.
- Rui Zhang et al. Lora-fa: Memory-efficient low-rank adaptation via feature re-alignment. *arXiv preprint arXiv:2302.05653*, 2023. URL <https://arxiv.org/abs/2302.05653>.
- Shangqian Zhao, Shiyu Li, and Yi Ma. Zero initialization: Initializing neural networks with zero-valued parameters. *arXiv preprint arXiv:2207.05848*, 2022. URL <https://arxiv.org/abs/2207.05848>.
- Tianshi Zhao, Zhen Sun, Xiaodong Wang, Fei Zhou, Yang Guo, and Alexander J Smola. Extending torchelastic for stateful training jobs. *arXiv preprint arXiv:2006.06873*, 2020.
- Yuwei Zhao, Yifan Zhang, et al. In-rank: Incremental low-rank learning. *arXiv preprint arXiv:2303.11246*, 2023. URL <https://arxiv.org/abs/2303.11246>.