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 (**???**).

To quantify this, consider a model with Ψ parameters. Storing these parameters and their gradients in 16-bit precision formats like FP16 or BF16 requires 2Ψ bytes each. Optimizer states are typically stored in 32-bit precision (FP32) for numerical stability, necessitating an additional 4Ψ bytes for each of the parameters, gradient momentum and variance, amounting to 12Ψ bytes. Therefore, the total memory requirement sums up to 16Ψ 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 18Ψ 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.

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 (**?**) and tensor parallelism (**?**) 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** (**?**), 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 (**?**), 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)** (**?**) further partitions optimizer states and gradients across data-parallel processes, eliminating redundancy and significantly reducing memory footprint. **Fully Sharded Data Parallel (FSDP)** (**?**) 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 (**?**), GPT-3/4 (**?**), Mistral (**?**), and Gopher (**?**) 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.

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 (**?**). 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) (**?**) *reparameterizes* a weight matrix $W \in \mathbb{R}^{m\times n}$ as:

$$W = W_0 + BA, \tag{1}$$

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 (**?**), 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 (**????**), 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 (**???**).

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 (**?**). 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 (**?**).

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.

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 (**???**). This intrinsic low-rank structure of gradients has been applied to reduce communication costs (**??**) and to decrease memory footprints during training (**???**).

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:

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

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 gradients 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.

Our Approach 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 Accelerating GaLore with Second-Order Information

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), 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 Problem Setup

Assume that the training/finetuning problem can be formulated as the following loss minimization problem:

$$\min_{\theta} \Phi(\theta)$$

where θ is the model parameter and Φ is the loss function. The gradient of the loss function with respect to the model parameter is denoted as $\mathbf{g} = \nabla_\theta \Phi(\theta)$. The goal is to find an optimal update direction δ that minimizes the loss function. Now in the case of GaLore, the update direction can be parameterized as: $\delta = \mathbf{P^T u}$, where $\mathbf{P}$ is the projection matrix, which means that the Taylor series expansion for that update is given by:

$$\Phi(\theta + \mathbf{P^T u}) \approx \Phi(\theta) + \mathbf{g^T P^T u} + \frac{1}{2}\mathbf{u^T P H P^T u}$$

where $\mathbf{H} = \nabla_\theta^2 \Phi(\theta)$ is the Hessian Matrix. Then the minimizer w.r.t. the low rank update direction $\mathbf{u}$ is given by:

$$\mathbf{u}^* = \arg\min_{u \in \mathbb{R}^r} \{\Phi(\theta_t) + \mathbf{g^T P^T u} + \frac{1}{2}\mathbf{u^T P H P^T u}\}$$

Algorithm 1: Pseudocode for **project** method

```
def project(full_rank_grad, iter):
    # Check if projection matrices need updating
    if core is None or iter % update_proj_gap == 0:
        core, factors = get_orthogonal_matrix(full_rank_grad, rank)
    # Transform gradient to low-rank space
    low_rank_grad = transform(factors, full_rank_grad)
    # Apply natural gradient transform
    low_rank_grad = natural_gradient_transform(low_rank_grad)
    return low_rank_grad
```

Then the gradient descent update is given by:

$$\theta_{t+1} = \theta_t + \mathbf{P^T}\mathbf{u}^*$$

.

In GaLore, the projection matrix $\mathbf{P}$ is chosen to be the top r singular vectors of the gradient matrix $\mathbf{g}$ and the low rank update direction is chosen using Adam or AdamW optimizer. This method is memory efficient as it only requires storing the projection matrix $\mathbf{P}$ and costly optimizer states, now only depend upon the low rank update direction $\mathbf{u}$.

However, the performance of GaLore is still not on par with the vanilla AdamW optimizer. To bridge this gap, we propose Natural GaLore, which incorporates second-order information in the optimization process.

2.2 Algorithm

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.

Step 1: Low-Rank Gradient Projection Given a full-rank gradient tensor $\mathbf{g} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, we project it onto a low-rank subspace using Tucker decomposition (**?**). The Tucker decomposition approximates $\mathbf{g}$ as:

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

where:

- $\mathcal{C} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_d}$ is the core tensor 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 $\{\mathbf{U}^{(i)}\}$:

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

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

Algorithm 2: Pseudocode for **natural_gradient_transform**

```
def natural_gradient_transform(low_rank_grad):
   # Flatten low-rank gradient to vector
   grad_vector = low_rank_grad.reshape(-1)
   # Update gradient history buffer
   append grad_vector to grad_history
   if len(grad_history) > history_size:
      remove oldest gradient from grad_history
   # Form matrix G from gradient history
   G = stack(grad_history)
   # Compute S = I + $\lambda^{-1}$ $G^\top G$
   S = (1 / lambda_damping) * ($G^\top$ @ G)
   add 1.0 to diagonal elements of S
   # Compute $G^\top$ grad_vector
   GTg = $G^\top$ @ grad_vector
   # Solve S z = GTg for z using Cholesky decomposition
   L = Cholesky_decompose(S)
   solve L u = GTg for u
   solve $L^\top$ z = u for z
   # Compute natural gradient
   G_z = G @ z
   ng_vector = (1 / lambda_damping) * grad_vector
   ng_vector -= (1 / lambda_damping$^2$) * G_z
   # Reshape back to original low-rank shape
   natural_grad = ng_vector.reshape_as(low_rank_grad)
   return natural_grad
```

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:

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

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

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

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

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 (**?**).

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:

Algorithm 3: Pseudocode for **project_back** methods

```
def project_back(low_rank_update):
    # Reconstruct full-rank update from low-rank update
    full_rank_update = inverse_transform(factors, low_rank_update)
    # Apply scaling if necessary
    return full_rank_update * scale
```

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

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 $\mathbf{Sz} = \mathbf{y}$ for $\mathbf{z}$.
3. Compute $\tilde{\mathbf{g}}_t = \dfrac{1}{\lambda}(\mathbf{g}_t - \mathbf{Gz})$.

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

Step 5: Efficient Solution Using Cholesky Decomposition To solve the linear system $\mathbf{Sz} = \mathbf{y}$ efficiently, we use **Cholesky decomposition**:

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

Step 6: Updating the Model Parameters Finally, we use the natural gradient $\tilde{\mathbf{g}}_t$ to update the model parameters in the low-rank subspace. The full-rank update can be reconstructed using the inverse transformation if necessary.

2.3 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.

2.4 Implementation Details

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 Numerical Stability

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

2.4.5 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** λ: Affects numerical stability and convergence; may require tuning for different models or datasets.

2.5 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.

2.6 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 (**?**). 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.

2.7 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 (η is the learning rate):

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

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 $\tilde{G}_t$:

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

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

$$\tilde{G}_t = M_t / \sqrt{V_t + \epsilon} \tag{6}$$

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.

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

$$G_t = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i W_t C_i) \tag{7}$$

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:

$$\mathrm{sr}(G_t) \leq \mathrm{sr}() + \left(\frac{1-\eta\lambda_2}{1-\eta\lambda_1}\right)^{2(t-t_0)} \frac{\|G_0 - \|_F^2}{\|\|_2^2} \tag{8}$$

where is the projection of G_{t_0} onto the minimal eigenspace $_1$ 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 $\mathrm{sr}()$, which is estimated to be low-rank by the following:

Corollary 3.2 (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 $\mathrm{sr}() \leq n - N'$ and thus $\mathrm{sr}(G_t) \leq n/2$ for large t.*

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.

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.

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 (**?**).

4.1 Combining with Existing Techniques

Natural GaLore is compatible with existing memory-efficient optimization techniques. For example, Natural GaLore can be combined with gradient checkpointing (**?**) to further reduce memory usage.

4.2 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.

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	✓	✓

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

5 Experiments

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

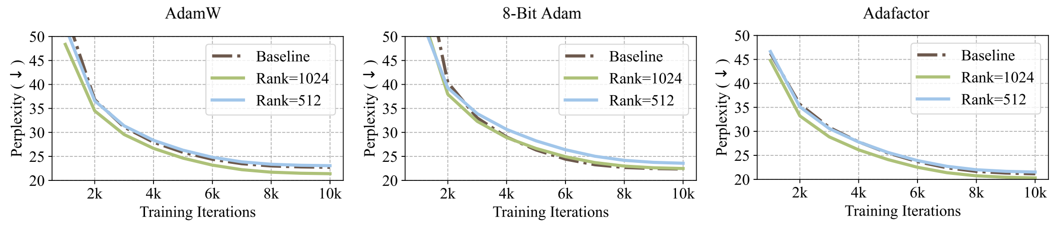

Figure 1: Applying Natural GaLore to different optimizers for pre-training LLaMA 1B on C4 dataset for 10K steps. Validation perplexity over training steps is reported. We apply Natural GaLore to each optimizer with the rank of 512 and 1024, where the 1B model dimension is 2048.

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.

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

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

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 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.

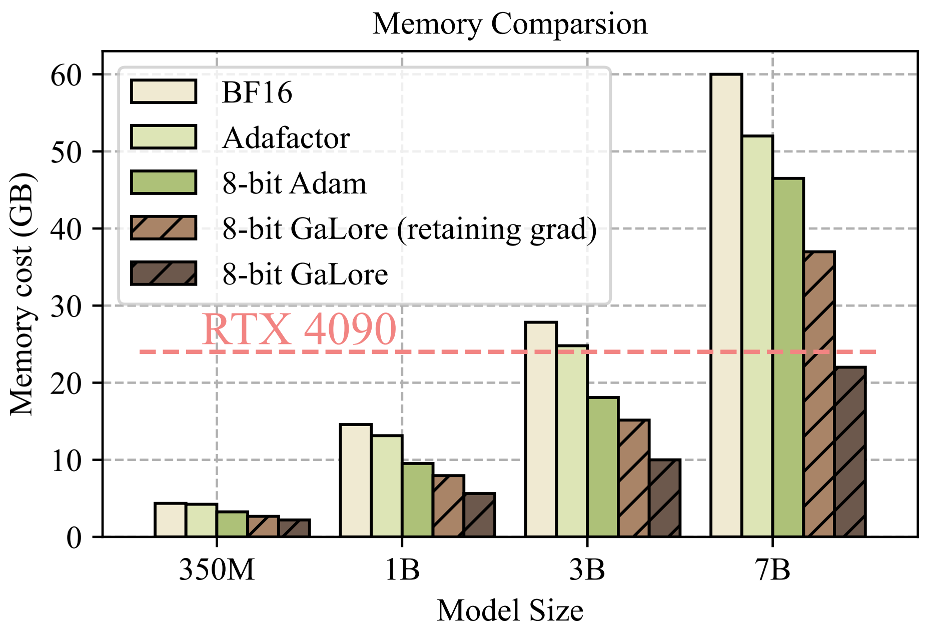

Figure 2: Memory usage for different methods at various model sizes, evaluated with a token batch size of 256. 8-bit Natural GaLore (retaining grad) disables per-layer weight updates but stores weight gradients during training.

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

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.

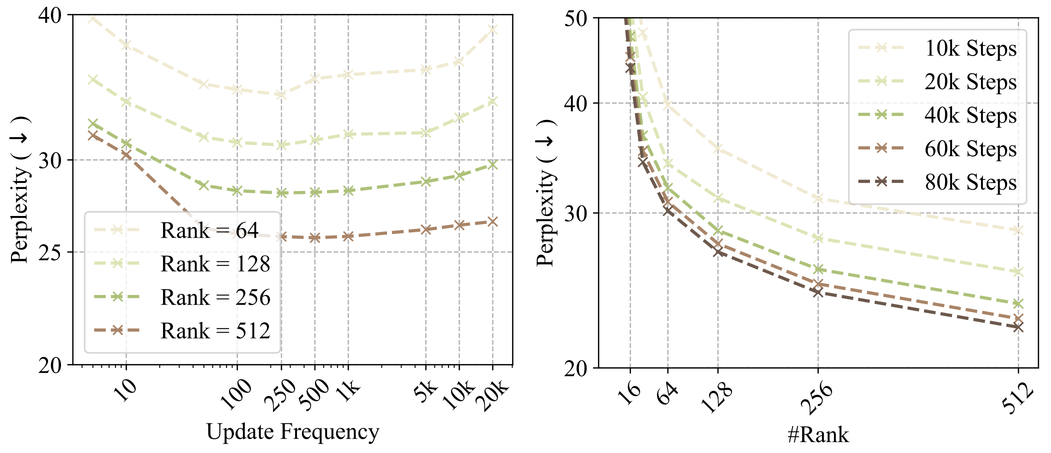

Figure 3: Ablation study of Natural GaLore on 130M models. **Left:** varying subspace update frequency T. **Right:** varying subspace rank and training iterations.

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.