Natural GaLore: A Memory Efficient Approach for LLM Training and Finetuning

Arijit Das

October 11, 2024

Outline

Introduction

AI Systems

- **Large Language Models (LLMs)** have achieved remarkable performance in various disciplines.
- However, turning LLMs into reliable AI systems remains challenging.
- Every AI system will make mistakes, but the monolithic nature of LLMs makes it difficult to understand and correct these errors.
- By building modular programs, using LLMs as specialized components, we can create more reliable and interpretable systems.

LLM Program

- A LLM program $\Phi : \mathcal{X} \rightarrow \mathcal{Y}$, is a function where input and output space $\mathcal{X}, \mathcal{Y}$ is natural language.
- Φ is assumed to make calls to modules $\mathcal{M} = \left(\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_{|\mathcal{M}|}\right)$ where each module $\mathcal{M}_i : \mathcal{X}_i \rightarrow \mathcal{Y}_i$ is a declarative LLM invocation, defined via inherently fuzzy natural language descriptions.

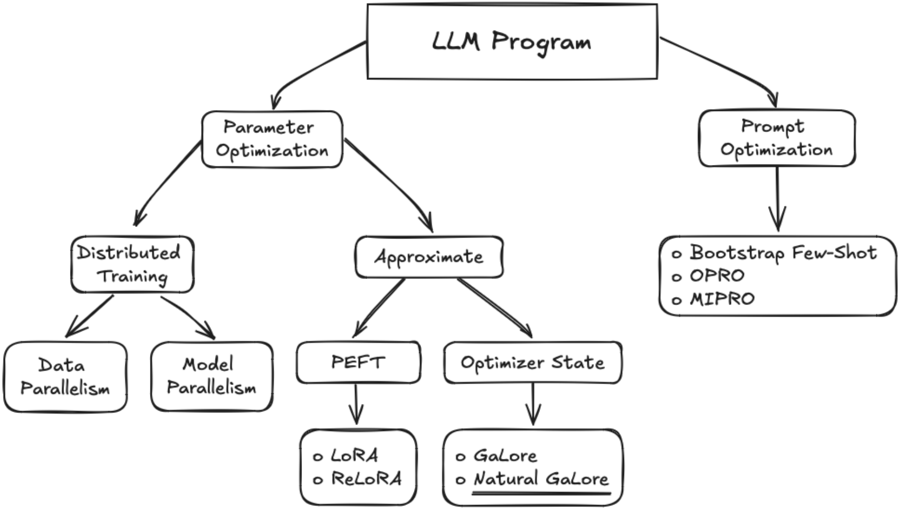
LLM Program Optimization

- The LLM program optimization problem is then defined by:

$$\arg\min_{\Pi,\Theta} \sum_{(x,y)\in(\mathcal{X}\times\mathcal{Y})} \mathcal{L}(\Phi_{\Theta,\Pi}(x), y)$$

  given a labeled dataset $(\mathcal{X} \times \mathcal{Y})$ and a loss function $\mathcal{L}$.
- For each module $\mathcal{M}_i$, the optimization problem determines:
  - **Prompt Optimization**: String prompt Π_i in which inputs $\mathcal{X}_i$ are plugged in.
  - **Parameter Optimization**: Weights Θ_i which are assigned to the module $\mathcal{M}_i$.

LLM Program Optimization



- In this work, I focus on the parameter optimization problem

Parameter Optimization

Next Token Prediction in LLMs

Generative LLMs predict the next token based on previously observed tokens (causal prediction).

$$\text{Prob}_\theta(x) = \prod_{t=1}^{T} \text{Prob}_\theta(x_t \mid x_{<t}) \tag{1}$$

where $x_{<t} = (x_1, x_2, \ldots, x_{t-1})$.

Objective: Negative Log-Likelihood (NLL)

- The training objective is to minimize the Negative Log-Likelihood (NLL):

$$\Phi(\theta) = -\sum_{t=1}^{T} \log \mathrm{Prob}_{\theta}(x_t \mid x_{<t}) \tag{2}$$

- Penalizes low probability assignments to correct tokens.
- High-dimensional, non-convex optimization problem.

Parameter Optimization

- Training and fine-tuning LLMs demand enormous computational resources and are highly memory-intensive.
- Memory requirements:
  - Stem from storing billions of parameters, gradients, and optimizer states.
  - For example pre-training a Llama-7B model, just the model requires 72GB of memory: 14GB for parameters and gradients each, 42GB for optimizer states and 2GB for activations.
  - Limit the ability to train large models on hardware with limited memory capacity.
  - Increase training costs and environmental impact.
- **Objective**: Develop a memory-efficient approach for training and fine-tuning LLMs without sacrificing performance.

Distributed Training Techniques

- **Data Parallelism**
  - DDP combines data parallelism with efficient gradient synchronization.
  - **Pros**: Efficient gradient updates, good scalability.
  - **Cons**: Memory bottlenecks persist when model size exceeds single GPU capacity.
- **Model Parallelism**
  - Partitions model across multiple devices.
  - Techniques: Pipeline parallelism (Huang et al., 2019), Tensor parallelism (Shoeybi et al., 2019), Fully Sharded Data Parallel (FSDP) (Zhao et al., 2020).
  - **Pros**: Allows training of models larger than a single GPU.
  - **Cons**: Communication overhead, complex implementation.

Distributed Training Techniques

- Data and Model Parallelism can be augmented with further techniques to reduce memory usage.
- **Gradient Checkpointing** (Chen et al., 2016)
  - Stores subset of activations during forward pass.
  - Recomputes activations during backward pass.
  - **Pros**: Reduces memory usage.
  - **Cons**: Increases computational overhead.
- **Memory Offloading**
  - Moves optimizer states and gradients to CPU memory.
  - Techniques: ZeRO-Offload (Rajbhandari et al., 2020).
  - **Pros**: Significant memory reduction.
  - **Cons**: Increased system complexity, operational costs.

Parameter-Efficient Fine-Tuning (PEFT)

- **LoRA (Low-Rank Adaptation)** (Hu et al., 2022)
  - Reparameterizes weight matrices using low-rank adapters

$$W = W_0 + BA, \quad B \in \mathbb{R}^{n\times r}, A \in \mathbb{R}^{r\times m} \tag{3}$$

  - **Pros**: Reduces trainable parameters, lowers memory usage
  - **Cons**: May not match full fine-tuning performance, especially on complex tasks (Xia et al., 2024).
- **ReLoRA** (Lialin & Schatz, 2023)
  - Extends LoRA for pre-training.
  - Periodically updates frozen weights using learned adapters.
  - **Pros**: Enables continual learning with lower memory.
  - **Cons**: Requires initial full-rank training phase.

GaLore: Gradient Low-Rank Approximation

- Exploits low-rank structure of gradients to approximate optimizer states (Zhao et al., 2024).
- Projects gradients $\mathbf{g} \in \mathbb{R}^{n \times m}$ into a low-rank form using Singular Value Decomposition: $\mathbf{g} = \mathbf{P}\Sigma\mathbf{Q}^T$

$$\mathbf{g}_{\text{low-rank}} = \mathbf{P}^T\mathbf{g}, \quad \mathbf{P} \in \mathbb{R}^{n \times r} \tag{4}$$

- **Pros**:
  - Significant memory reduction (up to 30% compared to LoRA).
  - Full-parameter updates, maintaining model capacity.
- **Cons**:
  - Performance may not match full optimizer state methods.
  - Low-rank approximation may not capture full optimization dynamics.

Proposed Approach

Natural GaLore: Our Approach

Algorithm 1: Natural GaLore, PyTorch-like

```
for weight in model.parameters():
    grad = weight.grad
    # original space -> low rank subspace + natural gradient transform
    lor_grad = project(grad)
    # update by Adam, AdamW, etc.
    lor_update = update(lor_grad)
    # low rank subspace -> original space
    update = project_back(lor_update)
    weight.data += update
```

- **Goal**: Accelerating convergence of GaLore to bridge the gap with AdamW.
- **Key Idea**: Incorporate second-order information using the Fisher Information Matrix (FIM) to estimate natural gradients.

Low-Rank Gradient Descent in GaLore

- GaLore restricts updates to an affine subspace:
$$\mathbf{u}_k \in \theta_k + \text{Range}(\mathbf{P}_k) \tag{5}$$
- Projection matrix $\mathbf{P}_k \in \mathbb{R}^{n \times r}$ computed using compact SVD of the gradient matrix.

Taylor Series Approximation of Loss

$$\Phi(\theta_k + \mathbf{P}_k \mathbf{u}_k) \approx \Phi(\theta_k) + \mathbf{g}_k^T \mathbf{u}_k + \frac{1}{2} \mathbf{u}_k^T \mathbf{H}_k \mathbf{u}_k \tag{6}$$

where:

- $\mathbf{g}_k = \mathbf{P}_k^T \nabla_\theta \Phi(\theta_k)$ is the projected gradient.
- $\mathbf{H}_k = \mathbf{P}_k^T \nabla_\theta^2 \Phi(\theta_k) \mathbf{P}_k$ is the local Hessian matrix.

Fisher Information Matrix (FIM) Approximation

$$\mathbf{F}_k = \mathbb{E}_{x \sim p_{\text{data}}}[\mathbf{H}_k] \tag{7}$$

- The FIM captures the curvature of the loss landscape.
- Empirically estimated by:

$$\hat{\mathbf{F}}_k = \frac{1}{h} \sum_{k=1}^{h} \mathbf{g}_k \mathbf{g}_k^T \tag{8}$$

Optimal Update Direction

$$\mathbf{u}_k^* = \hat{\mathbf{F}}_k^{-1}\mathbf{g}_k \tag{9}$$

- Minimizes the loss in the local neighborhood.
- Leads to the following update step:

$$\theta_{k+1} = \theta_k - \eta \mathbf{P}_k \mathbf{u}_k^* \tag{10}$$

Improving GaLore with Fisher Efficiency

- GaLore's performance can be improved by incorporating second-order information via FIM.
- Natural gradient descent is Fisher efficient and reduces variance in gradient updates.

$$\mathrm{Var}[\theta_k] = \frac{1}{mk}\mathbf{F}_k^{-1}(\theta_k^*) + \mathcal{O}\left(\frac{1}{k^2}\right) \tag{11}$$

- Smaller variance translates to faster convergence.

Woodbury Identity for Efficient Natural Gradient

$$(A + UBU^T)^{-1} = A^{-1} - A^{-1}U(B^{-1} + U^T A^{-1} U)^{-1} U^T A^{-1} \tag{12}$$

- Used to compute the inverse FIM efficiently.
- Allows memory-efficient computation of the natural gradient.

Natural Gradient Estimate

$$\tilde{\mathbf{g}}_k = \frac{1}{\lambda}\mathbf{g}_k - \frac{1}{\lambda} G(\lambda I + G^T G)^{-1} G^T \mathbf{g}_k \tag{13}$$

where:

- $G = [\text{vec}(\mathbf{g}_k), \ldots, \text{vec}(\mathbf{g}_{k-s})]$ is the gradient history.

Efficient Computation with Cholesky Decomposition

$$Sz = y, \quad S = I + \frac{1}{\lambda} G^T G \tag{14}$$

- Solved using Cholesky decomposition to compute the natural gradient in $\mathcal{O}(s^2)$ time.
- Enables scalable low-rank optimization.

Conclusion: Memory-Efficient Second-Order Optimization

- Natural GaLore combines low-rank projection with efficient second-order updates.
- Improves convergence and reduces memory consumption.
- Fisher efficiency allows faster optimization in high-dimensional, non-convex spaces.

Natural Gradient Transform

- **Method**:
  - Apply inverse FIM to low-rank gradients from GaLore.
  - Utilize Woodbury Identity and Cholesky Decomposition for efficient computation.
- **Benefits**:
  - Faster convergence.
  - Variance reduction in gradient estimates.
  - Better utilization of curvature information.

Natural Gradient Computation

- **Optimal Update Direction**:

$$\mathbf{u}_k^* = \hat{\mathbf{F}}_k^{-1}\mathbf{g}_k$$

where $\hat{\mathbf{F}}_k$ is the empirical Fisher Information Matrix.

- **Parameter Update**:

$$\theta_{k+1} = \theta_k - \eta \mathbf{P}_k \mathbf{u}_k^*$$

- **Efficient Inverse FIM Computation**:
  - Use Woodbury Identity:

$$(\lambda I + GG^T)^{-1} = \frac{1}{\lambda}I - \frac{1}{\lambda^2}G(I + \frac{1}{\lambda}G^T G)^{-1}G^T$$

  - Compute using Cholesky Decomposition and matrix-vector products.

Advantages of Our Approach

- **Incorporates Curvature Information**:
  - Accounts for the geometry of the loss landscape.
  - Enables more informed optimization steps.
- **Variance Reduction**:
  - Reduces variance in gradient estimates.
  - Leads to faster convergence, especially in limited iterations.
- **Memory Efficiency**:
  - Maintains low memory footprint.
  - Can be implemented without significant computational overhead.

Experiments and Results

Pre-training Experiments on C4 Dataset

- **Models**: LLaMA variants with 60M, 300M, and 1.1B parameters.
- **Dataset**: C4 dataset.
- **Metrics**: Validation perplexity, memory consumption.
- **Results**:
  - Our method achieves lower perplexity than GaLore across all model sizes.
  - Closer performance to full optimizer state methods.
  - Maintains significant memory savings.
- **Conclusion**: Incorporating natural gradients enhances performance without additional memory overhead.

Pre-training Experiments on C4 Dataset

Table: Comparison of *Natural GaLore* with other low-rank algorithms on pre-training various sizes of LLaMA models on the C4 dataset. Validation log perplexity is reported (averaged over 5 runs), along with a memory estimate (in gigabytes) of the total parameters and optimizer states based on BF16 format.

	60M	**130M**	**350M**	**1.1B**
Full-Rank	3.52 (0.36G)	3.22 (0.76G)	2.93 (2.06G)	2.72 (7.80G)
Natural GaLore	**3.53** (0.24G)	**3.22** (0.52G)	**2.93** (1.22G)	**2.80** (4.38G)
GaLore	3.56 (0.24G)	3.24 (0.52G)	2.95 (1.22G)	2.90 (4.38G)
Low-Rank	4.35 (0.26G)	3.82 (0.54G)	3.62 (1.08G)	4.96 (3.57G)
LoRA	3.55 (0.36G)	3.52 (0.80G)	3.24 (1.76G)	2.96 (6.17G)
ReLoRA	3.61 (0.36G)	3.38 (0.80G)	3.37 (1.76G)	2.91 (6.17G)
Rank r/d_{model}	128 / 256	256 / 768	256 / 1024	512 / 2048
Training Tokens	1.1B	2.2B	6.4B	13.1B

Fine-Tuning on GLUE Benchmark

- **Model**: RoBERTa-Base.
- **Benchmark**: GLUE tasks (CoLA, MRPC, STS-B, etc.).
- **Comparison** with LoRA and full fine-tuning.
- **Results**:
  - Comparable or better performance than LoRA.
  - Achieved an average score of 86.05, close to full fine-tuning baseline of 86.28.
  - Less memory consumption.
- **Conclusion**: Effective for memory-efficient fine-tuning without sacrificing accuracy.

Fine-Tuning on GLUE Benchmark

Table: Evaluating *Natural GaLore* for memory-efficient fine-tuning on the GLUE benchmark using pre-trained RoBERTa-Base. We report the average score of all tasks. Memory consumption is reported in millions of parameters (M).

	Memory	**CoLA**	**STS-B**	**MRPC**	**RTE**	**SST-2**	**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	61.50	**90.80**	**92.10**	**79.50**	**94.20**	**87.05**	**92.30**	91.15	**86.05**
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	61.70	**90.90**	**92.25**	**79.80**	**94.40**	**87.20**	**92.35**	**91.25**	**86.23**
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

Fine-Tuning for Function Calling in AAS

- **Task**: Function calling using TinyAgent framework.
- **Model**: TinyLlama 1.1B.
- **Dataset**: TinyAgent dataset with 40K examples.
- **Metrics**: Success rate in generating correct function-calling plans.
- **Results**:
  - Our method achieves a success rate of **83.09%**.
  - Outperforms 16-bit LoRA (80.06%) and GPT-4-turbo by 4%.
  - Uses **30% less memory**.
- **Conclusion**: Enhances performance of smaller models, making them competitive with larger models.

Fine-Tuning for Function Calling in AAS

Table: Latency, size, and success rate of TinyAgent models before and after quantization. Latency is the end-to-end latency of the function calling planner, including the prompt processing time and generation.

Model	Weight Precision	Latency (seconds)	Model Size (GB)	Success Rate (%)
GPT-3.5	Unknown	3.2	Unknown	65.04
GPT-4-Turbo	Unknown	3.9	Unknown	79.08
TinyAgent-1.1B	16-bit (*Natural GaLore*)	3.9	2.2	**83.09**
	16-bit (LoRA)	3.9	2.2	80.06
TinyAgent-7B	16-bit (Erdogan et al., 2024)	19.5	14.5	84.95

Conclusion

Conclusion

- Presented **Natural GaLore**, a memory-efficient approach for LLM training and fine-tuning.
- Enhanced GaLore by incorporating **natural gradients** for better performance.
- Achieved significant memory savings without sacrificing accuracy.
- Validated effectiveness through extensive experiments on pre-training and fine-tuning tasks.
- Demonstrated practical benefits in advanced agentic systems.

Future Work

- Explore **low-memory and structured projection matrices** for further memory efficiency.
- Conduct more extensive empirical evaluations on fine-tuning tasks in Advanced Agentic Systems.
- Inspire future research on **optimizer state approximation** for memory-efficient training.
- Make large-scale model training more accessible on consumer-grade hardware.

Thank You

Questions?

References I

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, 2016. URL `https://arxiv.org/abs/1604.06174`.

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. TinyAgent: Function calling at the edge. *arXiv preprint arXiv:2409.00608*, 2024. URL `https://arxiv.org/abs/2409.00608`.

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. URL `https://arxiv.org/abs/1811.06965`.

References II

Vladimir Lialin and Arthur Schatz. ReLoRA: Low-rank fine-tuning reloaded. *arXiv preprint arXiv:2307.09769*, 2023. URL https://arxiv.org/abs/2307.09769.

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.

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. URL https://arxiv.org/abs/1909.08053.

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. To appear.

References III

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024. URL `https://arxiv.org/abs/2403.03507`.

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. URL `https://arxiv.org/abs/2006.06873`.