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

Arijit Das

October 11, 2024

Outline

Introduction

Advanced Agentic Systems (AAS)

- Since GPT-3 (Brown et al., 2020), the concept of in-context learning, has made LLMs extremely flexible and powerful.
- This has opened the door to build AAS, which can respond to a wide variety of queries and solving complex problems, which may require a combination of: knowledge, reasoning, compute and action.
- However, building reliable AAS for practical applications remains challenging.

Optimizing AAS

- Solving such a complex problem requires systemically breaking it down into modules.
- Each module or agent is a call to an LLM which performs a subtask, requiring its own set of prompts and parameters.
- These modules can then be combined together as a directed graph, which solves the problem.
- Building realiable and robust AAS hence requires good prompt and parameter optimization techniques.

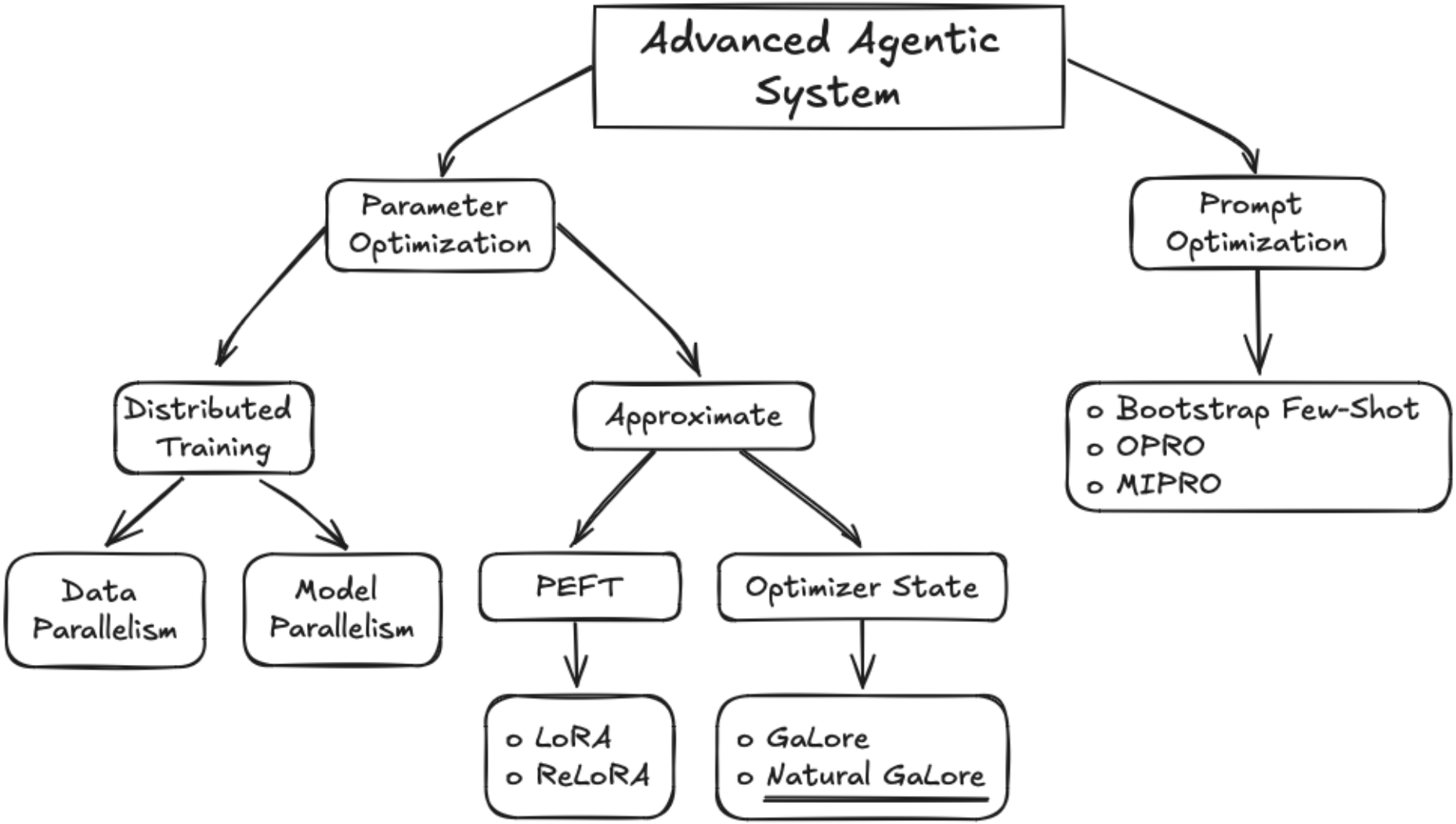
Prompt Optimization

- Prompt optimization is the process of finding the optimal prompt for a given LLM to maximize performance on a specific task.
  - **Prompt Proposal**: The challenge comes from the fact that they are inherently fuzzy and have a exponentially large solution space.
  - **Credit Assignment**: Another issue is to find efficient ways of inferring how each prompt variable contributes to performance.
- Algorithms like OPRO (Yang et al., 2024) and MIPRO (Opsahl-Ong et al., 2024), approach this problem by building efficient prompt proposal sampling, and credit assignment mechanisms following ideas from Bayesian optimization.

Parameter Optimization

- LLM training requires parameter optimization at three steps:
  - Pre-training: The model is optimized for the Next Token Prediction task on vast amounts of raw natural language text data.
  - Supervised Fine-tuning: Using high quality "Instruction Data", the model is transformed from essentially an autocomplete model, into one which can perform in context learning.
  - Preference optimization: Here responses are sampled, ranked according to user preferences and then the model is updated to prefer those.

AAS Optimization



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

Parameter Optimization

Next Token Prediction

- LLMs are trained to predict the next token based on previously observed tokens (causal prediction).
- Given tokens $x_{<t} = (x_1, x_2, \ldots, x_{t-1})$ the model is trained to maximize the probability of the next token x_t

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

Objective: Negative Log-Likelihood (NLL)

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

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

- Penalizes low probability assignments to correct tokens.
- However is a high-dimensional, non-convex optimization problem.

Adam Optimizer

- Adam (Kingma & Ba, 2014) is an iterative stochastic gradient descent algorithm with adaptive learning rates and momentum

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

where $\mathbf{g}_k = \nabla_\theta \Phi(\theta_k)$ and η is the learning rate.

- The optimal update direction is determined using the momentum term $\mathbf{m}_k \in \mathbb{R}^{r \times m}$ and the second moment estimate $\mathbf{v}_k \in \mathbb{R}^{r \times m}$. With all operations being elementwise, the update direction becomes:

$$\begin{aligned} \mathbf{m}_k &= \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\mathbf{g}_k & (3) \\ \mathbf{v}_k &= \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)\mathbf{g}_k^2 & (4) \\ \mathbf{u}_k^* &= \mathbf{m}_k / \sqrt{\mathbf{v}_k + \epsilon} & (5) \end{aligned}$$

Memory Challenges

- 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 for pre-training a Llama-7B model using Adam, requires 72GB of memory: 14GB for parameters and gradients each, 42GB for optimizer states and 2GB for activations.
  - This limits the ability to train large models on hardware with limited memory capacity.
  - Increases training costs and environmental impact.
- **Objective**: Develop a memory-efficient approach for training and fine-tuning LLMs without sacrificing performance.

Distributed Training Techniques

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.

Approximation Techniques

Parameter-Efficient Fine-Tuning (PEFT)

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

$$W = W_0 + BA, \quad B \in \mathbb{R}^{n \times r}, A \in \mathbb{R}^{r \times m} \tag{6}$$

  - **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).
- The gradients $\mathbf{g} \in \mathbb{R}^{n \times m}$ are projected onto a low-rank form using SVD: $\mathbf{g} = \mathbf{P}\Sigma\mathbf{Q}^T$

$$\mathbf{g}^{\text{low-rank}} = \mathbf{P}^T\mathbf{g}, \quad \mathbf{P} \in \mathbb{R}^{n \times r} \tag{7}$$

- Then after applying Adam to $\mathbf{g}^{\text{low-rank}}$ the full parameter update is reconstructed by applying $\mathbf{P}$.
- **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.

Natural GaLore

Natural GaLore: Proposed Approach

- **Goal**: Accelerating convergence of GaLore to bridge the gap with full state optimizers like Adam and AdamW.
- **Key Idea**: Incorporate second-order information into the low rank projected gradients using the Fisher Information Matrix (FIM) to efficiently estimate natural gradients.

Low-Rank Gradient Descent in GaLore

- GaLore restricts updates to an affine subspace defined by its principle components $\mathbf{P}_k \in \mathbb{R}^{n \times r}$:

$$\mathbf{u}_k \in \theta_k + \text{Range}(\mathbf{P}_k)$$

- Then the second-order Taylor series expansion of the loss function along this subspace is:

$$\Phi(\theta_k + \mathbf{P}_k \mathbf{u}_k) \approx \Phi(\theta_k) + \mathbf{g}_k^{\text{low-rank}\,T} \mathbf{u}_k + \frac{1}{2} \mathbf{u}_k^T \mathbf{H}_k \mathbf{u}_k$$

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

Fisher Information Matrix (FIM)

- **Key idea**: When the loss function can be written in terms of the KL-divergence, the Fisher Information Matrix (FIM) approximates the Hessian matrix:

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

- It captures the curvature of the loss landscape, enabling more efficient optimization.
- Natural gradient descent is Fisher efficient and reduces variance in gradient updates (Amari, 1998):

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

- Smaller variance translates to faster convergence.

Natural Gradient Transform

- In practice, FIM cannot be calculated efficiently.
- We approximate the empirical FIM using a mini-batch of $h \approx 20$ samples:

$$\hat{\mathbf{F}}_k = \frac{1}{h} \sum_{k=1}^{h} \mathbf{g}_k^{\text{low-rank}} \mathbf{g}_k^{\text{low-rank}\,T}$$

- The **natural gradient transform** can then be computed as:

$$\mathbf{g}_k^* = \hat{\mathbf{F}}_k^{-1} \mathbf{g}_k^{\text{low-rank}}$$

- As with GaLore, after applying Adam to $\mathbf{g}_k^*$ the full parameter update is reconstructed by applying $\mathbf{P}$.

Natural GaLore Algorithm

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
```

The algorithm combines low-rank projection from GaLore with efficient second-order updates.

Natural Gradient Computation

- Inverting the empirical FIM is computationally expensive.
- Here we use the Woodbury's Identity to compute the natural gradient efficiently:
  - By re-writing $\hat{\mathbf{F}}_k = \lambda I + GG^T$, we get:

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

  where $G = [\text{vec}(\mathbf{g}_k^{\text{low-rank}}), \ldots, \text{vec}(\mathbf{g}_{k-s}^{\text{low-rank}})]$ is the gradient history.
- Can be computed using Cholesky Decomposition and matrix-vector products in $\mathcal{O}(s^2)$ time:

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

- Hence enabling scalable low-rank second order optimization.

Advantages of Natural GaLore

- **Curvature Information**:
  - Accounts for the geometry of the loss landscape.
  - Enables more informed optimization steps in high-dimensional, non-convex spaces.
- **Variance Reduction**:
  - Natural gradients reduces variance in gradient estimates.
  - Leads to faster convergence, especially in limited iterations.
  - Further, when using a decaying learning rate schedule like with AdamW (Loshchilov & Hutter, 2017), the asymptotic convergence rate can be faster (Martens, 2020) by a significantly large constant factor.
- **Memory Efficiency**:
  - Maintains low memory footprint, as with GaLore.
  - Can be implemented without significant computational overhead.

Benchmarking Experiments

Pre-training Experiments on C4 Dataset

- **Models**: LLaMA variants with 60M, 130M, 350M, 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 accelerates convergence without significant 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, GaLore 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 signficantly 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

Application: Fine-Tuning TinyAgents

TinyAgent Framework

- TinyAgent framework (Erdogan et al., 2024) is an AAS for task-specific function-calling pipelines.
- **Goal**: Fulfill user queries through function-calling, respecting function interdependencies and passing correct arguments.
- **Key challenge**: Generating function-calling plans with correct functions, their arguments and interdependencies.

LLMCompiler for Function Calling

- LLMCompiler (Kim et al., 2023) generates function-calling plans with required functions and arguments.
- It can then compile the plan into executable sequences.
- **Key challenge**: Original paper, implmented with 7B+ models. Could we fine tune smaller language models to generate a function plan with correct syntax, argument values, and respect for dependencies.

Challenges with Off-the-Shelf TinyLlama 1.1B

- The off-the-shelf TinyLlama 1.1B (instruct-32k) performs poorly on function-calling tasks.
- Issues include:
  - Incorrect function sets.
  - Hallucinated function names.
  - Incorrect or missing dependencies.
  - Argument misplacement.

Innoculation by Fine-Tuning

- To solve this, we fine-tune TinyLlama 1.1B using the curated TinyAgent dataset (Erdogan et al., 2024).
- Fine-tuning enables the model to learn:
  - Task-specific patterns for function-calling.
  - Precise understanding of task dependencies and arguments.
- Now there are three possibilities after fine-tuning:
  - Performance improvement $\implies$ pre-training/instruction tuning data of the model was weak for this task.
  - No change in performance $\implies$ the model doesn't have the capacity to learn this task.
  - Otherwise, annotation artifact.

TinyAgent Dataset: Overview

- A meticulously curated dataset for building agentic systems on Apple MacBooks.
- Contains 40K examples of natural language queries and corresponding function-calling plans.
- Covers 16 distinct tasks such as:
  - Email, Contacts, SMS, Calendar, Notes, Reminders.
  - File Management, Zoom Meetings.
- Dataset split:
  - 38K training examples.
  - 1K validation examples.
  - 1K test examples.

Fine-Tuning Strategy

- During fine-tuning, the prompt includes:
  - Descriptions of the ground-truth functions.
  - Irrelevant functions serving as negative samples.
- The model learns to select the correct functions rather than simply memorizing the ground truth.
- Several **in-context examples** are used to demonstrate how queries translate into function-calling plans.
- Examples are selected via RAG, leveraging a **DeBERTa-v3-small** model (He et al., 2021) fine-tuned for multi-label classification to retrieve relevant examples from the training set.

Training Objective

- The training objective is to maximize the **accuracy of generated function-calling plans**.
- Success is defined as:
  - Correct function selection.
  - Correct arguments.
  - Correct order of function-calls.
- Verifying the selection of the correct function set is straightforward: set comparison.
- Ensuring correct argument handling and order requires constructing the associated **Directed Acyclic Graph (DAG)**.
- DAGs check for equality in the structure and dependencies of the function-calls.

Fine-Tuning for Function Calling

- **Task**: Function calling using TinyAgent framework.
- **Model**: TinyLlama 1.1B fine-tuned for 3 epochs, batch size 32, learning rate 7×10^{-5}.
- **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

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.

Questions?

References I

Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. URL `https://ieeexplore.ieee.org/document/6790500`.

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. URL `https://arxiv.org/abs/2005.14165`.

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`.

References II

Pengcheng He, Jianfeng Gao, and Weizhu Chen. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. *arXiv preprint arXiv:2111.09543*, 2021. URL https://arxiv.org/abs/2111.09543.

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.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023. URL https://arxiv.org/abs/2312.04511.

References III

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL https://arxiv.org/abs/1412.6980.

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

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. URL https://arxiv.org/abs/1711.05101.

James Martens. New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21:1–76, 2020. URL https://www.jmlr.org/papers/volume21/17-678/17-678.pdf.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs, 2024. URL https://arxiv.org/abs/2406.11695.

References IV

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.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

References V

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`.