MLorc: Momentum Low-rank Compression for Memory Efficient Large Language Model Adaptation

Wei Shen[1]
University of Virginia

Zhang Yaxiang [1,2]
National University of Singapore

Minhui Huang
Meta

Mengfan Xu
University of Massachusetts Amherst

Jiawei Zhang
University of Wisconsin-Madison

Cong Shen
University of Virginia

Abstract

With the increasing size of large language models (LLMs), full-parameter fine-tuning imposes substantial memory demands. To alleviate this, we propose a novel memory-efficient training paradigm called **M**omentum **Lo**w-**r**ank **c**ompression (MLorc). The key idea of MLorc is to compress and reconstruct the momentum of matrix parameters during training to reduce memory consumption. Compared to LoRA, MLorc avoids enforcing a fixed-rank constraint on weight update matrices and thus enables full-parameter learning. Compared to GaLore, MLorc directly compress the momentum rather than gradients, thereby better preserving the training dynamics of full-parameter fine-tuning. We provide a theoretical guarantee for its convergence under mild assumptions. Empirically, MLorc consistently outperforms other memory-efficient training methods, matches or even exceeds the performance of full fine-tuning at small ranks (e.g., $r = 4$), and generalizes well across different optimizers, all while not compromising time or memory efficiency.

[1]Equal contribution. The first two authors are listed in alphabetical order.
[2]Corresponding author. Address correspondance at e1353410@u.nus.edu.

Proceedings of the 29th International Conference on Artificial Intelligence and Statistics (AISTATS) 2026, Tangier, Morocco. PMLR: Volume 300. Copyright 2026 by the author(s).

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated strong generalization capabilities on downstream tasks after fine-tuning (Liu et al., 2019; Raffel et al., 2020; Li and Liang, 2021). However, full-parameter fine-tuning is prohibitively expensive in terms of GPU memory. In addition to storing billions of model parameters and activation values, training also requires memory for gradients and various optimizer states (e.g., first- and second-order momentum terms in Adam). Without memory-saving techniques, standard AdamW consumes approximately three times more memory for gradients and optimizer states than for storing model parameters alone.

One promising approach to reduce this memory overhead is to design memory-efficient optimization paradigms tailored for fine-tuning, where the objective is to adapt the model to specific tasks. LoRA (Low-Rank Adaptation) (Hu et al., 2022) is one of the most widely adopted parameter-efficient fine-tuning (PEFT) methods: it freezes the original model weights and introduces trainable, low-rank updates. However, LoRA inherently limits the space of possible weight updates due to its low-rank constraint, and its reparameterization can significantly alter training dynamics. Prior studies have shown that LoRA may underperform full-parameter fine-tuning on certain tasks (Biderman et al., 2024; Xia et al., 2024) and exhibit distinct update patterns (Liu et al., 2024b).

Recently, GaLore (Gradient Low-Rank Projection) (Zhao et al., 2024), another memory-efficient optimization approach, has garnered attention. GaLore projects gradients and optimizer states (e.g., momentum) into a low-dimensional subspace for storage and uses the same projector to reconstruct optimizer states used to update weight. The projectors are periodically updated through Singular Value Decomposition

(SVD) on stochastic gradient matrices. GaLore claims to overcome the limitations of low-rank methods like LoRA by canceling low-rank factorization and improving training dynamics. Nevertheless, both prior research (Luo et al., 2024) and our experiments reveal that GaLore usually underperforms, even compared to LoRA. We attribute GaLore's underperformance to its suboptimal training dynamics, that is, the reconstruction in GaLore can break the structure of the momentum, due to the instability of singular vectors of the stochastic gradient. We will analyze this in detail in Section 3.

To address these challenges, we propose a new memory-efficient training paradigm, **M**omentum **Lo**w-**r**ank **c**ompression (MLorc). Unlike GaLore, MLorc directly compresses and reconstructs momentum instead of gradients with Randomized SVD (RSVD) (Halko et al., 2011) and then uses these compressed momentum to run some benchmark optimizers like Adam (Diederik, 2014) or Lion (Chen et al., 2023), thereby maintaining closer alignment with the training dynamics of full-parameter fine-tuning. The motivation of designing MLorc stems from our empirical observation that, during LLM fine-tuning, the momentum of matrix parameters often exhibits an approximately low-rank structure, implying that compressing the momentum does not result in significant information loss. Moreover, directly compressing the momentum can avoid the affect of unstable stochastic gradient.

Our main contributions can be summarized as follows:

- We propose a new memory-efficient training paradigm called **M**omentum **Lo**w-**r**ank **c**ompression (MLorc). The key idea of MLorc is to compress and reconstruct the momentum of matrix parameters during training to reduce memory consumption. The main motivation of MLorc is from our empirical observation of approximately low-rank structure of matrix parameters' momentum.
- We provide a theoretical convergence guarantee for MLorc with the Lion optimizer (Chen et al., 2023), matching the original Lion's sample complexity (Dong et al., 2024) under mild assumptions.
- We validate the effectiveness of MLorc through extensive experiments across diverse models, datasets, and optimizers. Our results demonstrate that MLorc outperforms LoRA (Hu et al., 2022), GaLore (Zhao et al., 2024), and LDAdamW (Robert et al., 2024) on math and coding tasks with LLaMA2-7B, and achieves the highest average performance on GLUE tasks with RoBERTa-Base (Liu et al., 2019). At the same time, MLorc maintains competitive runtime and memory efficiency compared to other memory-saving methods.

2 RELATED WORK

Low-rank adaptation. Low-rank adaptation methods, such as LoRA (Hu et al., 2022), have been proposed to enhance the memory efficiency of fine-tuning large language models. LoRA introduces trainable low-rank matrices into each layer of a pre-trained model, significantly reducing the number of trainable parameters while maintaining performance comparable to full fine-tuning. Inspired by LoRA, Flora (Hao et al., 2024) periodically resamples random projection matrices during training to compress the gradients, aiming to achieve higher-rank updates over time while maintaining same level memory consumption. There are also other variants of LoRA designed for improving performance and other purposes (Meng et al., 2024; Kalajdzievski, 2023; Dettmers et al., 2023; Hayou et al., 2024; Zhang et al., 2023b; Li et al., 2024; Zi et al., 2023; Wang et al., 2023; Li et al., 2024; Zhang et al., 2023a).

In contrast, GaLore (Zhao et al., 2024) is a memory-efficient fine-tuning method that reduces the storage cost of gradients and optimizer states by projecting them into a dynamically learned low-rank subspace. There are also other variants of GaLore designed for improving time efficiency and further reducing memory footprint (Zhang et al., 2024; Rajabi et al., 2025; Yang et al., 2025). However, a recent study (He et al., 2024) shows that GaLore does not always converge to the optimal solution under standard assumptions. He et al. (2024) proposed GoLore, a variant of GaLore that utilized random low-rank projection instead of the greedy one used in GaLore. Recent studies have also introduced GaLore-inspired variants. For example, Fira (Chen et al., 2024) improves performance by combining the exact gradient with the GaLore update. LDAdam (Robert et al., 2024) incorporates a projection-aware update rule for optimizer states together with a generalized error-feedback mechanism, explicitly addressing the compression of both gradients and optimizer states.

Memory-efficient optimization. There are also other techniques to reduce memory footprint during training, including gradient checkpointing (Chen et al., 2016), quantization (Dettmers et al., 2022; Li et al., 2023) and other memory-efficient optimization methods (AdaLomo (Lv et al., 2023), MeZO (Malladi et al., 2023), etc). These methods address different aspects of the memory bottleneck. They are orthogonal to our methods and some of them can be combined with MLorc to further reduce the memory footprint.

Matrix compression. Matrix compression techniques, particularly those based on Singular Value Decomposition (SVD), play an important role in model

compression (Wang et al., 2025; Liu et al., 2024a) and reducing memory footprint (Zhao et al., 2024) in model training. Randomized SVD (RSVD) (Halko et al., 2011) is an efficient variant of SVD. SVD decomposes a matrix into low-rank components that preserve most of its information, while RSVD accelerates this process by approximating the dominant singular subspace using random projections. These methods enable compact representation of gradients and optimizer states, thus laying the foundation of our method.

3 PRELIMINARIES AND MLORC

In this section, we introduce our main method. We begin by presenting the preliminaries on which our method is built, including LoRA and GaLore. We then formally elaborate on MLorc, covering the algorithmic framework and steps, memory analysis, and convergence analysis.

3.1 Preliminaries

Notation. For a matrix $A \in \mathbb{R}^{m\times n}$, we denote its Frobenius norm as $\|A\|_F$, denote its entrywise l_1 norm as $\|A\|_{1,1} \triangleq \sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|$. Given a batch sample $B = \{\xi^i\}_{i=1}^{b}$, we denote $\nabla f(W;B) = \frac{1}{b}\sum_{i=1}^{b}\nabla f(W;\xi_i)$.

3.1.1 LoRA

LoRA (Hu et al., 2022) is a parameter-efficient fine-tuning technique designed for adapting large pre-trained models to downstream tasks. Instead of updating the full model weights, LoRA freezes the original parameters and injects trainable low-rank matrices into specific layers (typically attention or feedforward layers). This significantly reduces the number of trainable parameters and memory requirements during fine-tuning. Initial weight of the model $W_0 \in R^{m\times n}$ is frozen, and weight update is achieved by updating two low rank matrices: $B \in R^{m\times r}$ and $A \in R^{r\times n}$, typically $r \ll m, n$, as illustrated in the following formula:

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

Despite its memory efficiency, LoRA has several limitations. First, the imposed low-rank constraint can restrict the expressiveness of weight updates, potentially limiting performance on tasks that require more complex adaptations. Second, LoRA introduces a reparameterization of the weight update process, which alters the training dynamics and can lead to suboptimal convergence in some scenarios (Zhao et al., 2024; Meng et al., 2024). Empirical studies have shown that LoRA can underperform full fine-tuning on certain tasks (Biderman et al., 2024; Xia et al., 2024).

3.1.2 GaLore

GaLore (Zhao et al., 2024) is a recent memory-efficient training paradigm designed to reduce the memory footprint of optimizer states and gradients during fine-tuning of large language models. Unlike LoRA, which freezes the model and injects low-rank trainable adapters into the weight matrices, GaLore applies a low-rank projection directly to the gradients and optimizer states. Specifically, it performs periodic SVD on the gradients to identify a low-rank subspace, into which the optimizer states (e.g., momentum, variance) are projected. This strategy allows GaLore to maintain full-parameter weight updates while significantly compressing the memory required for training, aiming to preserve training dynamics more faithfully than LoRA's reparameterized updates.

However, there is still room for improvement in GaLore. Although GaLore does not constrain the weight updates themselves to be low-rank, it relies on fixed (over a certain number of steps) low-rank projections, which may still limit its ability to fully capture dynamic gradient information.

To be specific, in step t, GaLore (on Adam) first gets projector P_t: it is updated every T steps using the singular vector of gradient G_t; otherwise P_t is equal to P_{t-1}. Subsequently, GaLore projects G_t: $R_t = P_t^T G_t$ and first/second order momentum M_t, V_t is constructed by exponential average of R_t, just like original Adam. Finally, low-rank update $N_t = \frac{M_t}{\sqrt{V_t}+\epsilon}$ and GaLore uses P_t to project back N_t. To ensure GaLore's training dynamics align with those of full-parameter training, it implicitly depends on two key assumptions: (1) gradients exhibit a low-rank structure, and (2) the eigenspace of N_t can be properly recovered by pre-defined projectors.

While the first assumption is well-supported by prior studies (Zhao et al., 2022; Cosson et al., 2023), the second is questionable in the context of mini-batch training. Infrequent projector updates can result in misaligned projections and reconstructions, and even with expensive high-frequency updates, a critical limitation remains: **there exists no well-defined projector for back-projection of** N_t, since momentum is an accumulation (i.e., weighted average) of mini-batch gradients across different steps. For example, with default $\beta_2 = 0.999$, gradients of 100 steps earlier still have comparable weight with current gradients in second-order momentum; hence, M_t and V_t's eigen space is very different from $g_{t-\tau}$'s for any τ. Also, N_t is a non-linear transformation of M_t and V_t, which makes the preservation of eigenspace impossible. Consequently, N_t's eigenspace cannot be recovered from any single-step gradient's eigenspace. This motivates

us to shift focus from gradient compression to momentum compression – **directly compressing and reconstructing momentum rather than gradients**.

3.2 MLorc

3.2.1 Algorithm and Implementation

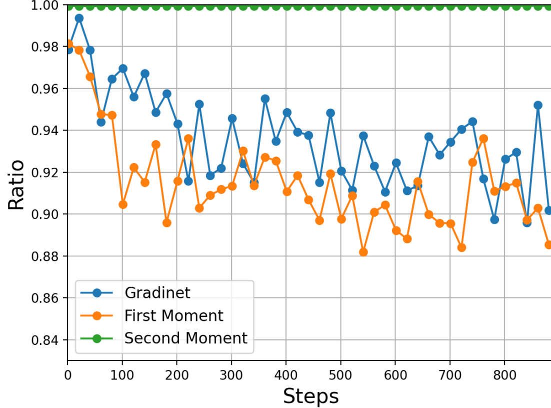


Figure 1: Ratio of top-8 singular values to total singular values for gradient, first moment, and second moment during AdamW finetuning of RoBERTa-base on the STSB dataset.

To enable compression at the momentum level, we first investigate whether momentum exhibit low-rank structure. We analyze various components involved in optimization by examining the concentration of singular values in gradients and momenta. For a matrix $A \in R^{m\times n}$ (suppose $m \geq n \geq 8$) with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, we calculate its ratio of top-8 singular values as $\frac{\sum_{i=1}^{8}\sigma_i}{\sum_{i=1}^{n}\sigma_i}$. We use this ratio to quantify the concentration of singular values (i.e., the 'low-rankness') of a matrix, where a larger value reflects a stronger low-rank structure. We compute its ratio for the gradient, first moment, and second moment matrices during AdamW fine-tuning of RoBERTa-base on the STSB dataset and get Figure 1. As illustrated in Figure 1, the first-order momentum shows a spectral pattern similar to that of the gradients, while the second-order momentum demonstrates an even stronger low-rank structure. More experimental evidence can be found in Appendix C.1. Motivated by these empirical observations, along with our earlier analysis of GaLore's training dynamics, we propose **M**omentum **Lo**w-**r**ank **c**ompression (MLorc), a new memory-efficient training paradigm for large-scale model fine-tuning.

The core idea of MLorc is to efficiently and accurately compress momentum for storage and reconstruct momentum to update weight, and we chose RSVD (Halko et al., 2011) to do this. A detailed introduction to RSVD is deferred to Appendix A. Here, we highlight a key property: the time complexity of RSVD is $O(mnr)$, which is on the same order as the projection and back-projection operations. Notably, MLorc can be applied to any optimizer (e.g, Adam, Lion) with momentum. Taking AdamW as an example: at each optimization step, we first reconstruct the first and second order momentum $\tilde{m}_{t-1}, \tilde{v}_{t-1}$ from the compressed optimizer state: $\tilde{m}_{t-1} = m_{u,t-1}m_{s,t-1}m_{v,t-1}^\top$, $\tilde{v}_{t-1} = v_{u,t-1}v_{s,t-1}v_{v,t-1}^\top$, update them using the current gradient: $m_t = \beta_1\tilde{m}_{t-1} + (1-\beta_1)g_t$, $v_t = \beta_2\tilde{v}_{t-1} + (1-\beta_2)g_t^2$, and then compress the updated momenta using RSVD: $(m_{u,t}, m_{s,t}, m_{v,t}) = \mathrm{RSVD}(m_t)$, $(v_{u,t}, v_{s,t}, v_{v,t}) = \mathrm{RSVD}(v_t)$. Finally, we use these updated momenta to perform the usual parameter update.

There is also a special consideration for second-order momentum, which must remain entry-wise nonnegative. A straightforward approach is to apply an entry-wise $ReLU$ to the reconstructed second-order momentum $\tilde{v}_{t-1}$. However, this introduces zeros in the reconstructed values, and since β_2 is typically set very close to 1, these zeros can result in extremely small values in the updated second-order momentum. This can destabilize training and degrade model performance. To address this, we add a small constant entry-wise to the zero values in $ReLU(\tilde{v}_{t-1})$. Given that parameter groups often have different scales, this constant should be chosen adaptively. In practice, we set it to the absolute mean of the negative part of the reconstructed momentum, which is usually much smaller than the positive part. For example, we first calculate the absolute mean of the negative part of the reconstructed momentum $\zeta(\tilde{v}_{t-1}) = \frac{1}{|\{i:(\tilde{v}_{t-1})_i<0\}|}\sum_{i:(\tilde{v}_{t-1})_i<0}|(\tilde{v}_{t-1})_i|$. Then, we update the $\tilde{v}_{t-1}$ according to the following formula:

$$\tilde{v}_{t-1} \leftarrow ReLU(\tilde{v}_{t-1}) + \zeta(\tilde{v}_{t-1}) \cdot \mathbf{1}_{\{\tilde{v}_{t-1}<0\}}, \quad (2)$$

where $\mathbf{1}_{\{\tilde{v}_{t-1}<0\}}$ is the indicator vector of the negative entries of $\tilde{v}_{t-1}$. This modification is different from adding ϵ on the square root of second-order momentum: 0s here come from the error introduced by momentum compression rather than the small magnitude of the corresponding gradient element.

Algorithm 1 shows the detailed description of MLorc-AdamW. Similarly, the high-level idea of MLorc can also be applied to other optimizers, such as Lion (Chen et al., 2023). Since Lion only maintains a single first-order momentum, MLorc-Lion is simpler than MLorc-AdamW. It only requires reconstructing the previous momentum $\tilde{m}_{t-1}$ at each step t and applying RSVD to compress the current momentum m_t. Algorithm 2

Table 1: Memory comparison. Assume $W \in R^{m \times n}$, rank r.

	Full finetuning (AdamW)	LoRA (AdamW)	GaLore	MLorc-AdamW
Weights	mn	$mn+mr+nr$	mn	mn
Optimizer States	$2mn$	$2mr+2nr$	$mr+2nr$	$2mr+2nr$

Algorithm 1 MLorc-AdamW

1: **Input**: Initial weights W_1, learning rate α, betas β_1, β_2, weight decay rate λ, constant ε, target rank r, oversampling parameter p, batch size b.
2: Initialize RSVD factors: $(m_{u,0}, m_{s,0}, m_{v,0}) \leftarrow 0$, $(v_{u,0}, v_{s,0}, v_{v,0}) \leftarrow 0$, $t \leftarrow 1$
3: **while** not converged **do**
4: Sample a mini-batch $B_t = \{\xi_t^i\}_{i=1}^b$ uniformly at random
5: Compute gradient: $g_t \leftarrow \nabla f(W_t; B_t)$
6: $\tilde{m}_{t-1} \leftarrow m_{u,t-1} m_{s,t-1} m_{v,t-1}^\top$
7: $\tilde{v}_{t-1} \leftarrow v_{u,t-1} v_{s,t-1} v_{v,t-1}^\top$
8: Update $\tilde{v}_{t-1}$ according to (2)
9: $m_t \leftarrow \beta_1 \tilde{m}_{t-1} + (1-\beta_1) g_t$
10: $v_t \leftarrow \beta_2 \tilde{v}_{t-1} + (1-\beta_2) g_t^2$
11: $(m_{u,t}, m_{s,t}, m_{v,t}) \leftarrow \mathrm{RSVD}(m_t, r, p)$
12: $(v_{u,t}, v_{s,t}, v_{v,t}) \leftarrow \mathrm{RSVD}(v_t, r, p)$
13: $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
14: $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
15: $W_{t+1} \leftarrow W_t - \alpha(\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon} + \lambda W_t)$
16: $t \leftarrow t+1$
17: **end while**
18: **return** W_t

Algorithm 2 MLorc-Lion

Input: Initial weights W_1, learning rate α, betas β_1, β_2, target rank r, oversampling parameter p, batch size b.
2: Initialize RSVD factors: $(m_{u,0}, m_{s,0}, m_{v,0}) \leftarrow 0$, $t \leftarrow 1$
while not converged **do**
4: Sample a mini-batch $B_t = \{\xi_t^i\}_{i=1}^b$ uniformly at random
Compute gradient: $g_t \leftarrow \nabla f(W_t; B_t)$
6: $\tilde{m}_{t-1} \leftarrow m_{u,t-1} m_{s,t-1} m_{v,t-1}^\top$
$c_t \leftarrow \beta_1 \cdot \tilde{m}_{t-1} + (1-\beta_1) \cdot g_t$
8: $m_t \leftarrow \beta_2 \cdot \tilde{m}_{t-1} + (1-\beta_2) \cdot g_t$
$(m_{u,t}, m_{s,t}, m_{v,t}) \leftarrow \mathrm{RSVD}(m_t, r, p)$
10: $W_{t+1} \leftarrow W_t - \alpha \cdot \mathrm{sign}(c_t)$
$t \leftarrow t+1$
12: **end while**
return W_t

shows a detailed description of MLorc-Lion. We will validate the effectiveness of MLorc across AdamW and Lion, and provide a theoretical convergence proof for MLorc-Lion.

3.2.2 Memory Consumption Analysis

In Table 1, we provide a comparison of the memory consumption of four methods: full finetuning, LoRA, GaLore, and MLorc. For a parameter matrix of size $m \times n$, full finetuning (with AdamW) needs to store the weight mn and two copies of the momentum terms $2mn$. LoRA requires storing additional weights $mr + nr$, but only needs $2mr + 2nr$ optimizer states. GaLore reduces optimizer memory to $2nr$; however, it also needs to store the projection matrix mr. MLorc-AdamW needs to store m_u, m_s, m_v and v_u, v_s, v_v, and since m_s and v_s can be absorbed into m_u, v_u (or into m_v, v_v), the optimizer memory of MLorc-AdamW is $2mr + 2nr$. Note that in typical fine-tuning settings (also in our experiments), $r \ll \min\{m, n\}$, which means, with the same rank r, LoRA, GaLore and MLorc achieve similar memory savings for the optimizer states.

GPU memory consumption during LLM training primarily comes from four sources: model weights, gradients, optimizer states, and activation values. Among these, weights and optimizer states always occupy GPU memory; therefore, in Table 1 we primarily compare these two components. Other memory overhead varies depending on the forward and backward phases as well as the specific implementation. LoRA typically needs to store gradients only for a small number of low-rank trainable parameters. For GaLore and MLorc, the gradients that must be stored depend on the specific implementation. Both GaLore and MLorc support per-layer weight updates (Lv et al., 2024), under which they need to store at most only one layer's weight gradients, thus consuming very little memory. As shown in Appendix C.2, under per-layer weight updates MLorc can use less memory than LoRA. Even without per-layer weight updates, although GaLore and MLorc require more memory than LoRA for storing gradients, the overall memory usage also depends on the activations, which in turn depends on batch size. In fact, in the experiments corresponding to Table 2 in our paper, the peak memory footprint often occurs during the forward pass, when all activation

values must be stored, indicating that gradients storage does not affect the overall memory peak. Taken together, these observations suggest that MLorc can indeed achieve memory efficiency comparable to that of LoRA and GaLore.

3.2.3 Convergence Analysis

In this section, we present a convergence analysis of MLorc-Lion (Algorithm 2), demonstrating that it can achieve the same sample complexity as the original Lion optimizer (Chen et al., 2023; Dong et al., 2024). Lion is a well-known optimizer and often achieves performance comparable to AdamW when optimizing neural networks. Lion utilizes the sign function to adjust the update magnitude of each component, which is conceptually similar to AdamW. In this work, we demonstrate the theoretical guarantees of our MLorc framework by presenting the convergence analysis of MLorc-Lion. The extension to MLorc-AdamW is left for future work. We consider optimizing loss function $f:\mathbb{R}^{m\times n}\to\mathbb{R}$. And we use the following standard assumptions.

Assumption 3.1. The loss function f is L-Lipschitz smooth, i.e. for any $W, W' \in \mathbb{R}^{m\times n}$, we have

$$\|\nabla f(W) - \nabla f(W')\|_F \le L\|W - W'\|_F.$$

Assumption 3.2. $\nabla f(W;\xi)$ is an unbiased stochastic estimator of the true gradient $\nabla f(W)$ and have a bounded variance, i.e.

$$\mathbb{E}[\nabla f(W;\xi)] = \nabla f(W)$$
$$\mathbb{E}\|\nabla f(W;\xi) - \nabla f(W)\|_F^2] \le \sigma^2.$$

With these standard assumptions, we have following theorem.

Theorem 3.3 (informal)**.** *Under Assumptions 3.1 and 3.2, applying Algorithm 2 with appropriate parameters, we have*

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(W_t)\|_{1,1}] \le O(1)\left[\frac{\sqrt{dL\Delta}}{\sqrt{T}} + \frac{\sigma\sqrt{d}}{\sqrt{b}}\right],$$

where $\Delta = f(W_1) - \inf_W f(W)$, $d = mn$.

The formal statement and proof of Theorem 3.3 can be found in Appendix B. According to Theorem 3.3, when $\sigma = 0$ (deterministic case), we can find an ϵ-entrywise ℓ_1-norm stationary point of f with a complexity of $O(\Delta L d\epsilon^{-2})$; when $\sigma \neq 0$ (stochastic case), with a large batch size $b = \Theta(d\sigma^2\epsilon^{-2})$, we can find an ϵ-entrywise ℓ_1-norm stationary point of f with a sample complexity of $O(\Delta L d^2\sigma^2\epsilon^{-4})$, matching the same sample complexity as the original Lion (Dong et al., 2024).

4 EXPERIMENTS

In this section, we demonstrate the effectiveness of MLorc through extensive experiments on NLG and NLU tasks, spanning diverse models, datasets, and optimizers.[3]

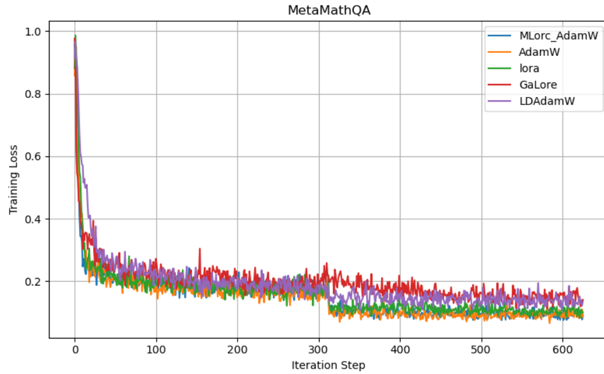
4.1 Experiments on NLG tasks with LLaMA2-7B

In this subsection, we evaluate MLorc's performance on large language models, focusing on NLG (Natural Language Generation) tasks. We fine-tuned LLaMA 2-7B (Touvron et al., 2023) on two tasks: math and code. To demonstrate the effectiveness of MLorc, We compare MLorc's performance with Full fine-tuning, LoRA (Hu et al., 2022), GaLore (Zhao et al., 2024) and its variant LDAdamW(Robert et al., 2024) (both optimized by AdamW). We also compare MLorc's performance on MLorc-Lion (Chen et al., 2023) with Full Lion and LoRA (Lion) to explore its applicability to different optimizers. Experimental results suggest that MLorc significantly reduces training loss during optimization and improves validation accuracy on test datasets.

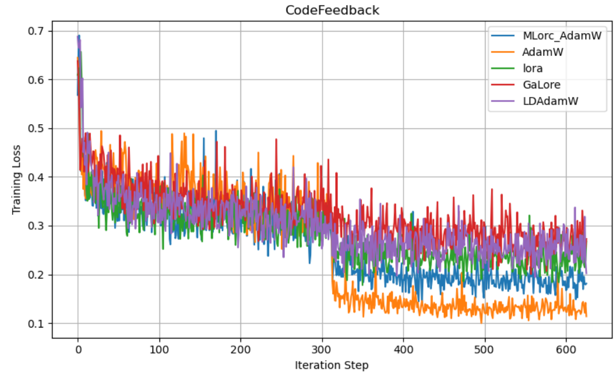
Experimental Setup. For the math task, the model was fine-tuned LLaMA 2-7B on the MetaMathQA dataset (Yu et al., 2023) and evaluated on GSM8K (Cobbe et al., 2021) validation sets. For the code task, the model was fine-tuned on the CodeFeedback dataset (Zheng et al., 2024) and evaluated on the HumanEval (Chen et al., 2021) dataset. All experiments were conducted on 1 H100-96 GPU, using subsets of training datasets containing 10K data points and were trained for 2 epochs. We use a rank=4 for all memory-efficient training paradigm, a batch size of 32 with gradient checkpointing and without gradient accumulation, a linear learning rate scheduler with a warmup ratio of 0.03. We set learning rate after tuning on each method and each dataset. It is worth mentioning that we set β_1 of MLorc-AdamW as 0.8 rather than the default value 0.9, in order to mitigate the influence of approximation error arising from RSVD. More hyperparameter setups, such as specific learning rates and oversampling parameters, can be found in Appendix D.1. Average accuracy over four evaluations and the standard deviation of accuracy is reported in Table 2.

As shown in Table 2, MLorc outperforms LoRA (Hu et al., 2022), GaLore (Zhao et al., 2024) and LDAdamW (Robert et al., 2024) on both math and coding task, significantly reduces accuracy gap between Full-parameter fine-tuning and existing

[3]Code is available at `https://github.com/weishen-git/MLorc`.



(a) Training Loss on MetaMathQA(Yu et al., 2023) dataset



(b) Training Loss on CodeFeedback(Zheng et al., 2024) dataset

Figure 2: Training Loss of AdamW of different methods

Table 2: Results of LLaMA 2-7B fine-tuned on math and code tasks. MLorc consistently outperforms other memory-efficient training paradigms. Lion version of GaLore and LDAdamW are not available; LoRA is independent of the choice of optimizer.

Method(r=4)	GSM8K	HumanEval
Full (AdamW)	$47.69_{\pm 0.15}$	$21.96_{\pm 0.46}$
MLorc (AdamW)	$47.37_{\pm 1.09}$	$20.70_{\pm 0.42}$
LoRA (AdamW)	$45.98_{\pm 0.52}$	$17.85_{\pm 1.07}$
GaLore	$38.89_{\pm 0.73}$	$17.25_{\pm 0.49}$
LDAdamW	$41.85_{\pm 0.60}$	$18.60_{\pm 1.08}$
Full (Lion)	$46.38_{\pm 1.11}$	$18.00_{\pm 0.30}$
MLorc (Lion)	$47.75_{\pm 0.25}$	$18.75_{\pm 0.78}$
LoRA (Lion)	$45.53_{\pm 0.52}$	$16.00_{\pm 0.83}$

memory-efficient training paradigms, demonstrating its ability in handling complex tasks on large language models. Additionally, the optimal learning rate of MLorc is much closer to Full-parameter fine-tuning than other memory-efficient training paradigms (see Appendix D), which suggests it might have similar training dynamics and indicates its utility in accelerating convergence.

Training Loss Curve. As shown in Figure 2 and 3, in most cases, the training loss of MLorc is smaller than other memory-efficient training paradigms, and is usually close to the full version, whenever in AdamW or Lion. This shows that MLorc behaves similarly to the full version, hence providing strong evidence for its effectiveness.

Time and Memory Efficiency. Table 3 compares memory consumption between different training methods. MLorc, GaLore and LoRA have similar memory efficiency; LDAdamW consumes more memory, probably due to its error feedback mechanism. Table 4 compares training time between different memory-efficient training methods. Experimental result shows that MLorc achieves time efficiency comparable to that of LoRA (Hu et al., 2022) and reduces training time compared to GaLore (Zhao et al., 2024), confirming that the additional overhead from compression and reconstruction is negligible in practical fine-tuning scenarios.

Table 3: Memory consumption of different methods with AdamW optimizer when training on MetaMathQA(Yu et al., 2023). Hyperparameters and other settings are same as previous experiments.

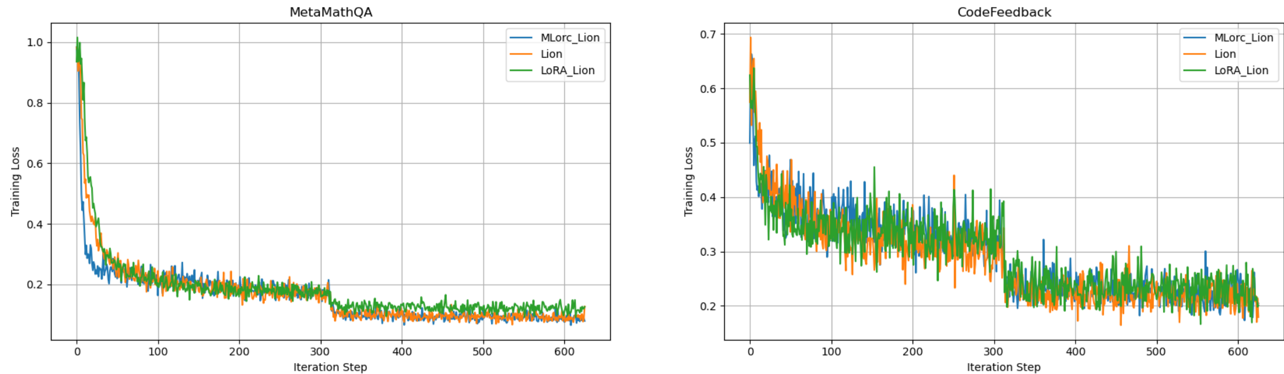
MLorc	LoRA	GaLore	LDAdamW
44.8GB	45.6GB	44.8GB	54.6GB

Table 4: Training time of different memory-efficient training methods with AdamW optimizer when training on MetaMathQA (Yu et al., 2023). Hyperparameters and other settings are the same as the previous experiments.

MLorc	LoRA	GaLore	LDAdamW
1h25min	1h24min	1h33min	1h26min

4.2 Experiments on Natural Language Understanding Tasks

In this subsection, we assess the effectiveness of MLorc in fine-tuning language models for natural language understanding (NLU) tasks. Specifically, we fine-tune pre-trained RoBERTa models (Liu et al., 2019) on the GLUE benchmark (Wang et al., 2018) using MLorc-AdamW, and compare its performance against full



(a) Training Loss on MetaMathQA(Yu et al., 2023) dataset

(b) Training Loss on CodeFeedback(Zheng et al., 2024) dataset

Figure 3: Training Loss of Full Lion and Lion with MLorc

Table 5: GLUE benchmark results of memory-efficient fine-tuning methods using pre-trained RoBERTa-Base. We set rank as 8 for all four memory-efficient methods. We used AdamW as the optimizer in full finetuning and LoRA. MLorc and Galore refer to MLorc-AdamW and Galore-AdamW respectively. Best performances among MLorc, Lora, GaLore and LDAdamW are highlighted in bold.

Method	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST2	STSB	Avg
Full	62.33	87.62	91.11	92.92	90.26	75.81	95.18	90.50	85.72
MLorc	**62.07**	**87.53**	**90.77**	**93.19**	88.99	**77.98**	**95.18**	90.59	**85.79**
LoRA	61.53	87.51	90.10	92.75	89.45	76.53	94.72	**90.74**	85.42
GaLore	60.34	86.84	90.10	92.42	88.15	71.12	94.38	90.50	84.23
LDAdamW	60.82	87.33	90.35	93.03	**90.06**	76.53	94.61	**90.74**	85.43

fine-tuning, LoRA (Hu et al., 2022), GaLore (Zhao et al., 2024) and LDAdamW (Robert et al., 2024). As shown in Table 5, MLorc significantly outperforms GaLore, surpasses LoRA and LDAdamW on most tasks, and achieves overall performance comparable to that of full fine-tuning. Detailed experimental settings can be found in Appendix D.2.

We also performed experiments to examine the low-rank structure of the gradient, first-order momentum, and second-order momentum during the full fine-tuning process of AdamW. The results on the STSB dataset are shown in Figure 1, and additional experimental results can be found in Appendix C.1.

5 CONCLUSIONS AND FUTURE WORK

In this work, we introduce MLorc (Momentum Low-rank Compression), a novel memory-efficient optimization paradigm designed to bridge the gap between parameter-efficient fine-tuning and full-parameter training for large language models (LLMs). Unlike existing low-rank adaptation methods such as LoRA (Hu et al., 2022) and GaLore (Zhao et al., 2024), MLorc leverages a previously underexplored insight: momentum is low-rank, and it can be compressed without significantly breaking training dynamics. By applying Randomized SVD (RSVD) (Halko et al., 2011) to compress and reconstruct momentum states instead of gradients, MLorc achieves a better balance between memory savings and training dynamics preservation.

Through comprehensive empirical evaluations across various model architectures, optimizers (e.g., AdamW and Lion), and NLP tasks (including NLG and NLU benchmarks), we demonstrate that MLorc: (1) consistently outperforms LoRA, GaLore and LDAdamW in terms of validation accuracy; (2) matches or exceeds full fine-tuning performance with a small compression rank (e.g., $r = 4$); (3) maintains comparable memory consumption to LoRA and achieves better time efficiency than GaLore; (4) shows strong training dynamics alignment with full fine-tuning, as evidenced by its training loss curves and optimal learning rates.

MLorc contributes to reducing the environmental impact of large-scale model training by significantly lowering GPU memory usage and computation overhead,

which can lead to reduced energy consumption during fine-tuning.

Looking ahead, we identify several promising directions for future work to validate and enhance MLorc: (1) Although our experiments focus on fine-tuning, extending MLorc to large-scale pre-training holds strong potential, as demonstrated by GaLore (Zhao et al., 2024), which shows that memory-efficient training schemes can significantly reduce memory usage while preserving model quality; (2) the current approach to compress momentum may not be optimal, so exploring alternative compression strategies with improved memory/time efficiency and approximation accuracy could be valuable; (3) our experiments were limited to models up to 7B parameters, so further empirical evaluation on larger-scale models (e.g., GPT-3 (Brown et al., 2020)) would help assess the scalability and effectiveness of MLorc.

Acknowledgments

The work of Jiawei Zhang was supported by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin–Madison with funding from the Wisconsin Alumni Research Foundation.

References

Biderman, D., Ortiz, J. J. G., Portes, J., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., et al. (2024). Lora learns less and forgets less. *CoRR*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Chen, X., Feng, K., Li, C., Lai, X., Yue, X., Yuan, Y., and Wang, G. (2024). Fira: Can we achieve full-rank training of llms under low-rank constraint? *arXiv preprint arXiv:2410.01623*.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., et al. (2023). Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. (2021). Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Cosson, R., Jadbabaie, A., Makur, A., Reisizadeh, A., and Shah, D. (2023). Low-rank gradient descent. *IEEE Open Journal of Control Systems*, 2:380–395.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. (2022). 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115.

Diederik, K. (2014). Adam: A method for stochastic optimization. *(No Title)*.

Dong, Y., Li, H., and Lin, Z. (2024). Convergence rate analysis of lion. *arXiv preprint arXiv:2411.07724*.

Halko, N., Martinsson, P.-G., and Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288.

Hao, Y., Cao, Y., and Mou, L. (2024). Flora: Low-rank adapters are secretly gradient compressors. In *International Conference on Machine Learning*, pages 17554–17571. PMLR.

Hayou, S., Ghosh, N., and Yu, B. (2024). Lora+: Efficient low rank adaptation of large models. In *International Conference on Machine Learning*, pages 17783–17806. PMLR.

He, Y., Li, P., Hu, Y., Chen, C., and Yuan, K. (2024). Subspace optimization for large language models with convergence guarantees. *arXiv preprint arXiv:2410.11289*.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. (2022). Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Kalajdzievski, D. (2023). A rank stabilization scaling factor for fine-tuning with lora. *arXiv preprint arXiv:2312.03732*.

Li, B., Chen, J., and Zhu, J. (2023). Memory efficient optimizers with 4-bit states. *Advances in Neural Information Processing Systems*, 36:15136–15171.

Li, D., Ma, Y., Wang, N., Ye, Z., Cheng, Z., Tang, Y., Zhang, Y., Duan, L., Zuo, J., Yang, C., et al. (2024). Mixlora: Enhancing large language models fine-tuning with lora-based mixture of experts. *arXiv preprint arXiv:2404.15159*.

Li, X. L. and Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597.

Liu, S.-Y., Khadkevich, M., Fung, N. C., Sakr, C., Yang, C.-H. H., Wang, C.-Y., Muralidharan, S., Yin, H., Cheng, K.-T., Kautz, J., et al. (2024a). Eora: Training-free compensation for compressed llm with eigenspace low-rank approximation. *arXiv preprint arXiv:2410.21271*.

Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., and Chen, M.-H. (2024b). Dora: Weight-decomposed low-rank adaptation. In *International Conference on Machine Learning*, pages 32100–32121. PMLR.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Luo, Q., Yu, H., and Li, X. (2024). Badam: A memory efficient full parameter optimization method for large language models. *Advances in Neural Information Processing Systems*, 37:24926–24958.

Lv, K., Yan, H., Guo, Q., Lv, H., and Qiu, X. (2023). Adalomo: Low-memory optimization with adaptive learning rate. *CoRR*.

Lv, K., Yang, Y., Liu, T., Guo, Q., and Qiu, X. (2024). Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8187–8198.

Malladi, S., Gao, T., Nichani, E., Damian, A., Lee, J. D., Chen, D., and Arora, S. (2023). Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075.

Meng, F., Wang, Z., and Zhang, M. (2024). Pissa: Principal singular values and singular vectors adaptation of large language models. *Advances in Neural Information Processing Systems*, 37:121038–121072.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Rajabi, S., Nonta, N., and Rambhatla, S. (2025). Subtrack your grad: Gradient subspace tracking for memory and time efficient full-parameter llm training. *arXiv preprint arXiv:2502.01586*.

Robert, T., Safaryan, M., Modoranu, I.-V., and Alistarh, D. (2024). Ldadam: Adaptive optimization from low-dimensional gradient statistics. *arXiv preprint arXiv:2410.16103*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355.

Wang, Q., Ke, J., Tomizuka, M., Chen, Y., Keutzer, K., and Xu, C. (2025). Dobi-svd: Differentiable svd for llm compression and some new perspectives. *arXiv preprint arXiv:2502.02723*.

Wang, Y., Lin, Y., Zeng, X., and Zhang, G. (2023). Multilora: Democratizing lora for better multi-task learning. *arXiv preprint arXiv:2311.11501*.

Xia, W., Qin, C., and Hazan, E. (2024). Chain of lora: Efficient fine-tuning of language models via residual learning. In *ICML 2024 Workshop on LLMs and Cognition*.

Yang, D. H., Amiri, M. M., Pedapati, T., Chaudhury, S., and Chen, P.-Y. (2025). Sparse gradient compression for fine-tuning large language models. *arXiv preprint arXiv:2502.00311*.

Yu, L., Jiang, W., Shi, H., Yu, J., Liu, Z., Zhang, Y., Kwok, J. T., Li, Z., Weller, A., and Liu, W. (2023). Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

Zhang, L., Zhang, L., Shi, S., Chu, X., and Li, B. (2023a). Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv:2308.03303*.

Zhang, Q., Chen, M., Bukharin, A., Karampatziakis, N., He, P., Cheng, Y., Chen, W., and Zhao, T. (2023b). Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*.

Zhang, Z., Jaiswal, A., Yin, L., Liu, S., Zhao, J., Tian, Y., and Wang, Z. (2024). Q-galore: Quantized galore with int4 projection and layer-adaptive low-rank gradients. *arXiv preprint arXiv:2407.08296*.

Zhao, J., Schaefer, F. T., and Anandkumar, A. (2022). Zero initialization: Initializing neural networks with only zeros and ones. *Transactions on Machine Learning Research*.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. (2024). Galore: Memory-efficient llm training by gradient low-rank projection. In *International Conference on Machine Learning*, pages 61121–61143. PMLR.

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. (2024). Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12834–12859.

Zi, B., Qi, X., Wang, L., Wang, J., Wong, K.-F., and Zhang, L. (2023). Delta-lora: Fine-tuning high-rank parameters with the delta of low-rank matrices. *arXiv preprint arXiv:2309.02411*.

Checklist

The checklist follows the references. For each question, choose your answer from the three possible options: Yes, No, Not Applicable. You are encouraged to include a justification to your answer, either by referencing the appropriate section of your paper or providing a brief inline description (1-2 sentences). Please do not modify the questions. Note that the Checklist section does not count towards the page limit. Not including the checklist in the first submission won't result in desk rejection, although in such case we will ask you to upload it during the author response period and include it in camera ready (if accepted).

In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes] We clearly describe the assumptions in Assumption 3.1, 3.2; algorithms in Algorithm 1, 2.
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes] We analysis the memory consumption in Section 3.2.2, and provide empirical memory consumption and training time of algorithms in Table 3 and 4.
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes] We provide anonymized source code in the Supplementary Material.

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [Yes] We clearly state the assumptions in Assumption 3.1, 3.2.
   (b) Complete proofs of all theoretical results. [Yes] We provide complete proofs in Appendix B.
   (c) Clear explanations of any assumptions. [Yes] We explain the assumptions in Section 3.2.3.

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Yes] We provide the code and instructions needed to reproduce the main experimental results in the Supplementary Material.
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes] We mentioned those details in Section 4 and Appendix C.
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Yes] We mentioned the details of error bars in Section 4: it's the standard deviation of four evaluation.
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes] We mentioned the computing infrastructure used in Section 4 and Appendix C.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. [Yes] We cited the existing assets we used.
   (b) The license information of the assets, if applicable. [Yes] We provide the license information of the assets in Appendix D.3.
   (c) New assets either in the supplemental material or as a URL, if applicable. [Yes] We upload our source code in the Supplementary Material.
   (d) Information about consent from data providers/curators. [Not Applicable]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]
5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [Not Applicable]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

Appendix

Appendix is organized as follows. Appendix A introduces details on RSVD. Appendix B provides a complete proof of Theorem 3.3. Appendix C presents additional experimental evidence on the low-rank structure and memory efficiency of MLorc. Appendix D gives detailed hyperparameter settings of experiments in Section 4 for reproducibility.

A Details on RSVD

Randomized Singular Value Decomposition (RSVD) is an efficient algorithm for computing a low-rank approximation of large matrices. Unlike the classical SVD, which can be computationally expensive for large-scale data, RSVD uses random projections to reduce the dimensionality of the input matrix before performing decomposition. This significantly accelerates the computation while retaining high approximation accuracy.

Algorithm 3 Randomized SVD (RSVD) with Oversampling

Require: Matrix $A \in \mathbb{R}^{m\times n}$, target rank r, oversampling parameter p
Ensure: Approximate rank-r SVD: $A \approx U\Sigma V^\top$
  $l \leftarrow r + p$
  Generate a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times l}$
  $Y \leftarrow A\Omega \in \mathbb{R}^{m\times l}$
  Compute the QR decomposition: $Y = QR$
  $B \leftarrow Q^\top A \in \mathbb{R}^{l\times n}$
  Compute SVD of the small matrix: $\tilde{U}, \Sigma, V^\top = SVD(B)$
  $U \leftarrow Q\tilde{U}$
  return U, Σ, V

The key idea behind RSVD is to first project the original matrix $A \in \mathbb{R}^{m\times n}$ onto a lower-dimensional subspace using a random matrix Ω producing a smaller matrix $Y = A\Omega$. Then, it performs a standard SVD or QR decomposition on Y, and reconstructs the approximate SVD A from this compressed representation.

Concerning the precision of RSVD, we have the following theorem, which is Theorem 10.5 of (Halko et al., 2011):

Lemma A.1 (Approximation error bound of RSVD)**.** *Let $A \in \mathbb{R}^{m\times n}$ have singular values $\sigma_1 \geq \sigma_2 \geq \cdots$. For a target rank $r \geq 2$ and oversampling parameter $p \geq 2$. When $r + p \leq \min\{m, n\}$, the randomized SVD algorithm produces an approximation A_{RS} such that*

$$\mathbb{E}\left[\|A - A_{RS}\|_F\right] \leq \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}} \left(\sum_{j>r} \sigma_j^2\right)^{\frac{1}{2}}. \tag{3}$$

This lemma suggests that when the desired rank r is small (e.g, $r = 4$), with a proper oversampling parameter p, RSVD has the same level approximation error(in expectation) as exact SVD up to a constant. We note that though the oversampling parameter p is involved in providing an error bound for RSVD, empirically, it does not significantly influence the experimental result. To reduce computational overhead, we set oversampling parameter $p = 0$ in each of our experiments.

B Proofs

Our proof generally follows that of (Dong et al., 2024), while additionally analyzing and bounding the error introduced by the low-rank compression in MLorc.

B.1 Lemmas

Lemma B.1. *When $r \geq 2, p \geq 2$ and $r + p \leq \min\{m, n\}$, define $\gamma = \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}}$. Then we have*

$$\mathbb{E}[\|\tilde{m}_t - m_t\|_F] \leq \gamma \|g_t\|_F$$

Proof. Let $\sigma_1 \geq \sigma_2 \geq \cdots$ be the singular values of m_t. We have

$$
\begin{aligned}
\mathbb{E}[\|\tilde{m}_t - m_t\|_F] &\leq \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}} (\sum_{j>r} \sigma_j^2)^{\frac{1}{2}} \\
&\leq \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}} \|m_t - \beta_2 \tilde{m}_{t-1}\|_F \\
&= \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}} (1-\beta_2)\|g_t\|_F,
\end{aligned}
$$

where the first inequality is due to Lemma A.1, while the second inequality follows from the Eckart–Young–Mirsky theorem, noting that $\tilde{m}_{t-1}$ is a rank r matrix. □

Lemma B.2. *Denote $\delta_t = c_t - \nabla f(W_t)$. Under Assumption 3.1, 3.2, we have*

$$
\begin{aligned}
\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\left[\|\delta_t\|_{1,1}\right] \leq & \frac{\sqrt{d}\sigma}{\sqrt{b}T(1-\beta_2)} + \frac{2L\alpha d}{1-\beta_2} + (|\beta_1 - \beta_2| + (1-\beta_1)) \cdot \frac{\sqrt{d}\sigma}{\sqrt{b(1-\beta_2)}} \\
& + \frac{1}{T}\sum_{t=1}^{T-1} \beta_1 \sqrt{d}\gamma \mathbb{E}\left[\|\nabla f(W_t)\|_{1,1}\right] + \gamma\frac{\sigma\sqrt{d}}{\sqrt{b}}
\end{aligned}
$$

Proof. Denote $\xi_t = g_t - \nabla f(W_t)$. We have

$$
\begin{aligned}
\delta_t =& \beta_1 \tilde{m}_{t-1} + (1-\beta_1) g_t - \nabla f(W_t) \\
=& \beta_1(\tilde{m}_{t-1} - m_{t-1}) + \beta_1\beta_2\tilde{m}_{t-2} + \beta_1(1-\beta_2)\nabla f(W_{t-1}) + (1-\beta_1)g_t - \nabla f(W_t) \\
=& \beta_1(\tilde{m}_{t-1} - m_{t-1}) + \beta_2(c_{t-1} - (1-\beta_1)\nabla f(W_{t-1})) + \beta_1(1-\beta_2)\nabla f(W_{t-1}) + (1-\beta_1)g_t - \nabla f(W_t) \\
=& \beta_1(\tilde{m}_{t-1} - m_{t-1}) + \beta_2(\delta_{t-1} + \nabla f(W_{t-1})) - \beta_2(1-\beta_1)g_{t-1} + \beta_1(1-\beta_2)\nabla f(W_{t-1}) + (1-\beta_1)g_t - \nabla f(W_t) \\
=& \beta_1(\tilde{m}_{t-1} - m_{t-1}) + \beta_2(\delta_{t-1} + \nabla f(W_{t-1})) + (\beta_1 - \beta_2)(\xi_{t-1} + \nabla f(W_{t-1})) + (1-\beta_1)(\xi_t + \nabla f(W_t)) - \nabla f(W_t) \\
=& \beta_1(\tilde{m}_{t-1} - m_{t-1}) + \beta_2\delta_{t-1} - \beta_1(\nabla f(W_t) - \nabla f(W_{t-1})) + (\beta_1 - \beta_2)\xi_{t-1} + (1-\beta_1)\xi_t \\
=& \beta_2^{t-1}\delta_1 + \sum_{k=2}^{k} \beta_2^{t-k} \Bigg(- \beta_1\left(\nabla f(W_k) - \nabla f(W_{k-1})\right) + (\beta_1 - \beta_2)\xi_{k-1} + (1-\beta_1)\xi_k + \beta_1(\tilde{m}_{k-1} - m_{k-1}) \Bigg) \\
=& \beta_2^{t-1}\delta_1 - \beta_1 \sum_{k=2}^{k} \beta_2^{t-k}\left(\nabla f(W_k) - \nabla f(W_{k-1})\right) + (\beta_1 - \beta_2)\sum_{k=2}^{t} \beta_2^{t-k}\xi_{k-1} \\
& + (1-\beta_1)\sum_{k=2}^{t} \beta_2^{t-k}\xi_k + \beta_1 \sum_{k=2}^{t} \beta_2^{t-k}(\tilde{m}_{k-1} - m_{k-1}).
\end{aligned}
$$

Taking expectations, and according to Lemma B.2, we have

$$
\begin{aligned}
\mathbb{E}\left[\|\delta_t\|_{1,1}\right] \leq & \sqrt{d}\Bigg\{ \beta_2^{t-1}\mathbb{E}\left[\|\delta_1\|_F\right] + \underbrace{\beta_1 \sum_{k=2}^{t} \beta_2^{t-k}\mathbb{E}\left[\|\nabla f(W_k) - \nabla f(W_{k-1})\|_F\right]}_{\text{term (a)}} \\
& + \underbrace{\mathbb{E}\left[\left\|(\beta_1 - \beta_2)\sum_{k=2}^{t}\beta_2^{t-k}\xi_{k-1} + (1-\beta_1)\sum_{k=2}^{t}\beta_2^{t-k}\xi_k\right\|_F\right]}_{\text{term (b)}} \\
& + \underbrace{\beta_1 \sum_{k=2}^{t}\beta_2^{t-k}\mathbb{E}\left[\|\tilde{m}_{k-1} - m_{k-1}\|_F\right]}_{\text{term (c)}} \Bigg\}.
\end{aligned}
$$

For term (a), we have

$$\begin{aligned}
\text{term (a)} \leq & L \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|W_k - W_{k-1}\|_F\right] \\
= & L\alpha \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\mathrm{sign}(c^{t-1})\right] \\
\leq & 2L\alpha\sqrt{d} \sum_{k=2}^{t} \beta_2^{t-k} \\
\leq & \frac{2L\alpha\sqrt{d}}{1-\beta_2}.
\end{aligned}$$

For term (b), we have

$$\begin{aligned}
\text{term (b)} &\leq |\beta_1 - \beta_2| \mathbb{E}\left[\left\|\sum_{k=2}^{t} \beta_2^{t-k} \xi_{k-1}\right\|_F\right] + (1-\beta_1)\mathbb{E}\left[\left\|\sum_{k=2}^{t} \beta_2^{t-k} \xi_k\right\|_F\right] \\
&\leq |\beta_1 - \beta_2| \sqrt{\mathbb{E}\left[\left\|\sum_{k=2}^{t} \beta_2^{t-k} \xi_{k-1}\right\|_F^2\right]} + (1-\beta_1)\sqrt{\mathbb{E}\left[\left\|\sum_{k=2}^{t} \beta_2^{t-k} \xi_k\right\|_F^2\right]} \\
&= |\beta_1 - \beta_2| \sqrt{\sum_{k=2}^{t} \beta_2^{2(t-k)} \mathbb{E}\left[\|\xi_{k-1}\|_F^2\right]} + (1-\beta_1)\sqrt{\sum_{k=2}^{t} \beta_2^{2(t-k)} \mathbb{E}\left[\|\xi_k\|_F^2\right]} \\
&= |\beta_1 - \beta_2| \sqrt{\sigma^2 \sum_{k=2}^{t} \beta_2^{2(t-k)}/b} + (1-\beta_1)\sqrt{\sigma^2 \sum_{k=2}^{t} \beta_2^{2(t-k)}/b} \\
&\leq (|\beta_1 - \beta_2| + (1-\beta_1)) \cdot \frac{\sigma}{\sqrt{b(1-\beta_2^2)}} \\
&\leq (|\beta_1 - \beta_2| + (1-\beta_1)) \cdot \frac{\sigma}{\sqrt{b(1-\beta_2)}}.
\end{aligned}$$

For term (c), according to Lemma B.1, we have

$$\begin{aligned}
\text{term (c)} \leq & \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\tilde{m}_{k-1} - m_{k-1}\|_F\right] \\
\leq & \gamma(1-\beta_2) \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|g_{k-1}\|_F\right] \\
\leq & \gamma(1-\beta_2) \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\nabla f(W_{k-1})\|_F\right] + \gamma(1-\beta_2) \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\xi_{k-1}\|_F\right] \\
\leq & \gamma(1-\beta_2) \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\nabla f(W_{k-1})\|_{1,1}\right] + \gamma \frac{\sigma}{\sqrt{b}}.
\end{aligned}$$

Plugging terms (a), (b), (c) back, we get

$$\begin{aligned}
\mathbb{E}\left[\|\delta_t\|_{1,1}\right] \leq & \sqrt{d}\{\beta_2^{k-1} \mathbb{E}\left[\|\delta_1\|_F\right] + \frac{2\beta_1 L\alpha\sqrt{d}}{1-\beta_2} + (|\beta_1 - \beta_2| + (1-\beta_1)) \cdot \frac{\sigma}{\sqrt{b(1-\beta_2)}}\} \\
& + \sqrt{d}\gamma(1-\beta_2)\beta_1 \sum_{k=2}^{t} \beta_2^{t-k} \mathbb{E}\left[\|\nabla f(W_{k-1})\|_{1,1}\right] + \gamma \frac{\sigma\sqrt{d}}{\sqrt{b}}
\end{aligned}$$

Initializing $m_0 = g_1$, we have $\mathbb{E}\left[\|\delta_1\|_F\right] = \mathbb{E}\left[\|g_1 - \nabla f(W_1)\|_F\right] \leq \sigma/\sqrt{b}$, and

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\|\delta_t\|_{1,1}\right] \leq & \frac{\sqrt{d}\sigma}{\sqrt{b}T(1-\beta_2)} + \frac{2L\alpha d}{1-\beta_2} + (|\beta_1-\beta_2| + (1-\beta_1)) \cdot \frac{\sqrt{d}\sigma}{\sqrt{b(1-\beta_2)}} \\ & + \frac{1}{T}\sum_{t=1}^{T-1}\beta_1\sqrt{d}\gamma\mathbb{E}\left[\|\nabla f(W_t)\|_{1,1}\right] + \gamma\frac{\sigma\sqrt{d}}{\sqrt{b}}\end{aligned}$$

□

B.2 Proof of Theorem 3.3

Formal statement of Theorem 3.3: Under Assumptions 3.1 and 3.2, applying Algorithm 2 with $r \geq 2, p \geq 2$, $r + p \leq \min\{m, n\}$, $\beta_1 \leq \frac{1}{4\gamma\sqrt{d}}$, we have

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\|\nabla f(W_t)\|_{1,1}\right] \leq O(1)\Bigg[& \frac{f(W_1) - f(W_T)}{\alpha T} + \frac{\sigma\sqrt{d}}{\sqrt{b}T(1-\beta_2)} \\ & + \frac{2L\alpha\beta_1 d}{1-\beta_2} + dL\alpha + \gamma\frac{\sigma\sqrt{d}}{\sqrt{b}} + (|\beta_1-\beta_2| + 1 - \beta_1) \cdot \frac{\sqrt{d}\sigma}{\sqrt{b(1-\beta_2)}}\Bigg].\end{aligned}$$

Denote $f(W_1) - \inf_W f(W) = \Delta$. Set $\gamma = \left(1 + \frac{r}{p-1}\right)^{\frac{1}{2}} = O(1)$, $\beta_2 = O(1)$, $\alpha = \sqrt{\frac{\Delta}{LdT}}$. We have

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(W_t)\|_{1,1}] \leq O(1)\left[\frac{\sqrt{dL\Delta}}{\sqrt{T}} + \frac{\sigma\sqrt{d}}{\sqrt{b}}\right].$$

Proof. According to Assumption 3.1, we have

$$\begin{aligned}& f(W_{t+1}) - f(W_t) \\ \leq & \langle\nabla f(W_t), W_{t+1} - W_t\rangle + \frac{L}{2}\|W_{t+1} - W_t\|_F^2 \\ = & -\alpha\langle\nabla f(W_t), \mathrm{sign}(c_t)\rangle + \frac{L\alpha^2}{2}\|\mathrm{sign}(c_t)\|_F^2 \\ = & -\alpha\langle\nabla f(W_t), \mathrm{sign}(\nabla f(W_t))\rangle - \alpha\langle\nabla f(W_t), \mathrm{sign}(c_t) - \mathrm{sign}(\nabla f(W_t))\rangle + \frac{L\alpha^2}{2}\|\mathrm{sign}(c_t)\|_F^2 \\ \leq & -\alpha\|\nabla f(W_t)\|_{1,1} + 2\alpha\|\delta_t\|_{1,1} + \frac{dL\alpha^2}{2}\end{aligned}$$

Taking expectations, and according to Lemma B.2, we have

$$\begin{aligned}& \mathbb{E}\left[f(W_T) - f(W_0)\right] \\ \leq & -\alpha\sum_{t=0}^{T-1}\mathbb{E}[\|\nabla f(W_t)\|_{1,1}] + 2\alpha\sum_{k=1}^{T}\mathbb{E}\left[\|\delta_t\|_{1,1}\right] + TdL\alpha^2/2 \\ \leq & -\alpha\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(W_t)\|_{1,1}] + 2\alpha T\Bigg\{\frac{\sqrt{d}\sigma}{\sqrt{b}T(1-\beta_2)} + \frac{2L\alpha\beta_1 d}{1-\beta_2} + (|\beta_1-\beta_2| + 1 - \beta_1) \cdot \frac{\sqrt{d}\sigma}{\sqrt{b(1-\beta_2)}} \\ & + \frac{1}{T}\sum_{t=1}^{T-1}\beta_1\sqrt{d}\gamma\mathbb{E}\left[\|\nabla f(W_t)\|_{1,1}\right] + \gamma\frac{\sigma\sqrt{d}}{\sqrt{b}}\Bigg\} + TdL\alpha^2/2.\end{aligned}$$

When $\beta_1 \leq \frac{1}{4\gamma\sqrt{d}}$, we have

$$1 - 2\beta_1\sqrt{d}\gamma \geq \frac{1}{2},$$

and

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left[\|\nabla f(W_t)\|_{1,1}\right] \le O(1)\Bigg[\frac{f(W_1)-f(W_T)}{\alpha T} + \frac{\sigma\sqrt{d}}{\sqrt{b}T(1-\beta_2)}$$

$$+ \frac{2L\alpha\beta_1 d}{1-\beta_2} + dL\alpha + \gamma\frac{\sigma\sqrt{d}}{\sqrt{b}} + (|\beta_1-\beta_2|+1-\beta_1)\cdot\frac{\sqrt{d}\sigma}{\sqrt{b(1-\beta_2)}}\Bigg],$$

Denote $f(W_1) - \inf_W f(W) = \Delta$. Set $\gamma = \left(1+\frac{r}{p-1}\right)^{\frac{1}{2}} = O(1)$, $\beta_2 = O(1)$, $\alpha = \sqrt{\frac{\Delta}{LdT}}$. We have

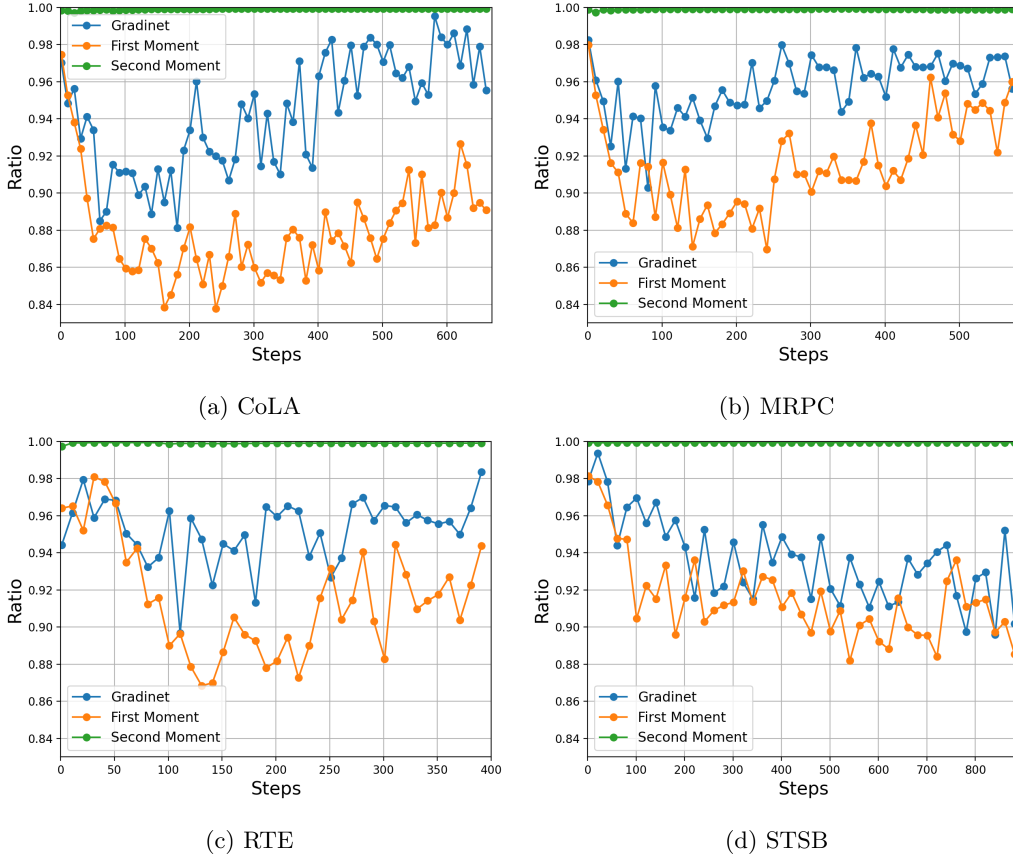
$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(W_t)\|_{1,1}] \le O(1)\left[\frac{\sqrt{dL\Delta}}{\sqrt{T}} + \frac{\sigma\sqrt{d}}{\sqrt{b}}\right].$$

Thus, when $\sigma = 0$ in deterministic case, we can find an ϵ-entrywise ℓ_1-norm stationary point of f with a complexity of $O(\Delta Ld\epsilon^{-2})$; when $\sigma \neq 0$ in stochastic case, with a large batch size $b = \Theta(d\sigma^2\epsilon^{-2})$, we can find an ϵ-entrywise ℓ_1-norm stationary point of f with a sample complexity of $O(\Delta Ld^2\sigma^2\epsilon^{-4})$, matching the same sample complexity as the original Lion (Dong et al., 2024).

□

C Additional Experimental Results

C.1 Low-rank Structures of the Gradients and Momenta



(a) CoLA (b) MRPC

(c) RTE (d) STSB

Figure 4: Ratio of top-8 singular values to total singular values for gradient, first moment, and second moment during AdamW finetuning of RoBERTa-base on the CoLA, MRPC, RTE, STSB datasets.

We conduct experiments examining the concentration of singular values in gradients and momenta during AdamW finetuning of RoBERTa-base on the CoLA, MRPC, RTE and STSB datasets. We set the batch size as 128, epochs as 20, learning rate as 1e-4 for all these four datasets. We use AdamW finetune the matrix parameters of query, key, value, output weights in attention layers and the intermediate and output weights in feed-forward layers. We conducted these experiments on NVIDIA RTX 6000 Ada GPUs. The average ratios of top-8 singular values to total singular values for gradient, first moment, and second moment of all these matrix parameters are reported in Figure 4. We can note that, in general, the momenta on all these datasets have highly concentrated singular values and exhibit low-rank structures, which is aligned with the intuition of our method.

C.2 Memory Footprint with Per-layer Weight Updates

As mentioned in Section 3.2, we can avoid storing full gradients in MLorc by using per-layer weight updates (Lv et al., 2024). Here we compare the memory consumption of LoRA and MLorc with per-layer weight updates with a batch size of 4.

Table 6: Memory footprint of MLorc with per-layer weight updates and LoRA with a batch size of 4. Apart from batch size, other hyperparameters and settings are same as previous experiments.

MLorc (per-layer update)	LoRA
16.8GB	17.7GB

Table 6 suggests that MLorc can even be more memory-efficient than LoRA with per-layer weight updates, which supports our claim in Section 3.2.

C.3 Ablation study

In MLorc-AdamW, we compress both the first moment and the second moment. In principle, we could instead compress only the first moment while retaining the original AdamW update rule for the second moment (denoted as MLorc_m in Table 7), or compress only the second moment while retaining the original AdamW update rule for the first moment (denoted as MLorc_v in Table 7). We conducted additional ablation experiments comparing MLorc_m, MLorc_v, and MLorc-AdamW . We adopted the same experimental settings as in the main paper, and the results are shown in Table 7. We observe that the performance differences among MLorc_m, MLorc_v, and MLorc-AdamW are relatively small. MLorc achieves better results than both MLorc_m and MLorc_v on four out of the eight datasets. However, MLorc requires significantly less memory than MLorc_m and MLorc_v. For example, in the MRPC experiment, full fine-tuning consumes 2498 MB, while MLorc_m and MLorc_v require 2027 MB and 2026 MB, respectively. In contrast, MLorc-AdamW only uses 1703 MB. Therefore, we believe that MLorc-AdamW is more practical when memory resources are limited.

Table 7: Ablation study on compressing different momenta in MLorc-AdamW

Method	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST2	STSB	Avg
Full	62.33	87.62	91.11	92.92	90.26	75.81	95.18	90.50	85.72
MLorc-AdamW	**62.07**	87.53	90.77	**93.19**	88.99	77.98	**95.18**	**90.59**	**85.79**
MLorc_m (Only compress m)	61.07	87.51	**91.34**	93.19	88.99	**78.70**	95.18	90.59	85.69
MLorc_v (Only compress v)	59.14	**87.67**	91.31	92.38	**90.32**	75.81	95.07	90.58	85.29

D Detailed Experimental Settings

D.1 Fine-Tuning on MetaMathQA and CodeFeedback

The pre-trained LLaMA2-7B model is from Hugging Face[4]. We have reported our batch size, epoch and other settings in Section 4.1. Also, for all methods, on GSM8K dataset, the max sequence length is 512; on Code-Feedback dataset, the max sequence length is 1024; the weight decay is 0. For GaLore, the subspace update

[4] https://huggingface.co/meta-llama/Llama-2-7b-chat-hf

frequency T is set to 300 on both datasets. Oversampling parameter p is set as 0 for MLorc on both datasets. The temperature for evaluation is 0.8 for math task and 0.1 for coding task, since a high temperature would lead to highly unstable performance on HumanEval dataset. For each method and each dataset, the learning rate is individually tuned. We present specific learning rates in Table 8.

Table 8: Learning rates of different methods when fine-tuning LLaMA2-7B on MetaMathQA and CodeFeedback dataset.

	MLorc-AdamW	Full (AdamW)	LoRA (AdamW)	GaLore	LDAdamW
MetaMathQA	7E-05	4E-05	1E-03	3E-03	3E-04
CodeFeedback	7E-05	9E-05	3E-04	2E-03	3E-04

	MLorc-Lion	Full (Lion)	LoRA (Lion)
MetaMathQA	1E-05	3E-05	1E-04
CodeFeedback	7E-06	2E-05	2E-04

D.2 Fine-Tuning on GLUE

The pre-trained RoBERTa-Base model is from Hugging Face[5]. We use the same batch size, number of epochs, and maximum sequence length across all methods, including Full fine-tuning, MLorc, LoRA, and GaLore. For each method and each dataset, the learning rate is individually tuned. The LoRA scaling factor α is set to 16 for all tasks. For GaLore, the subspace update frequency T is set to 50 for CoLA, MRPC, RTE, and STSB, and 100 for SST2, QNLI, MNLI, and QQP. We set the oversampling parameter p as 0 for MLorc for all datasets. Experiments for CoLA, MRPC, and RTE are conducted on NVIDIA H100 GPUs; STSB, SST2, and QNLI are conducted on NVIDIA RTX A6000 GPUs; MNLI on NVIDIA RTX 6000 Ada GPUs; and QQP on NVIDIA GeForce RTX 3090 GPUs. Detailed hyperparameter settings are provided in Table 9.

Table 9: Hyperparameter settings for the GLUE tasks. "LR" denotes the learning rate.

	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST2	STSB
Batch Size	128	128	128	128	128	128	128	128
Epochs	10	5	20	5	5	10	10	20
Max Seq. Len.	64	256	64	256	256	256	128	128
LR of Full	3E-05	3E-05	7E-05	1E-05	7E-05	3E-05	7E-06	1E-04
LR of MLorc	3E-05	1E-04	7E-05	5E-05	7E-05	5E-05	5E-05	7E-05
LR of LoRA	1E-03	3E-04	1E-03	5E-04	5E-04	7E-04	3E-04	5E-04
LR of GaLore	3E-04	3E-04	5E-04	3E-04	3E-04	5E-04	3E-04	3E-04
LR of LDAdamW	7E-05	7E-05	3E-04	7E-05	1E-04	5E-04	7E-05	1E-04

D.3 License information

LLaMA 2-7B (Touvron et al., 2023) is licensed under the LLaMA 2 Community License Agreement. CodeFeedback (Zheng et al., 2024) is licensed under Apache License 2.0. RoBERTa (Liu et al., 2019), MetaMathQA (Yu et al., 2023), GSM8K (Cobbe et al., 2021), HumanEval (Chen et al., 2021) are licensed under the MIT License.

[5]https://huggingface.co/docs/transformers/model_doc/roberta