GoRA: Gradient-driven Adaptive Low Rank Adaptation

Haonan He[1,2,†], **Peng Ye**[3,4,5,†], **Yuchen Ren**[3,6], **Yuan Yuan**[2], **Luyang Zhou**[7], **Shucun Ju**[7], **Lei Chen**[2*]

[1]University of Science and Technology of China
[2]Institute of Intelligent Machines, HFIPS, Chinese Academy of Sciences
[3]Shanghai Artificial Intelligence Laboratory [4]Fudan University
[5]The Chinese University of Hong [6]University of Sydney
[7]Anhui Disaster Warning & Agrometeorological Information Center

Abstract

Low-Rank Adaptation (LoRA) is a crucial method for efficiently fine-tuning large language models (LLMs), with its effectiveness influenced by two key factors: rank selection and weight initialization. While numerous LoRA variants have been proposed to improve performance by addressing one of these aspects, they often compromise usability or computational efficiency. In this paper, we analyze and identify the core limitations of existing approaches and propose a novel framework—**GoRA** (**G**radient-driven Adaptive L**o**w **R**ank **A**daptation)—that simultaneously adapts both the rank and initialization strategy within a unified framework. GoRA leverages gradient information during training to dynamically assign optimal ranks and initialize low-rank adapter weights in an adaptive manner. To our knowledge, GoRA is the first method that not only addresses the limitations of prior approaches—which often focus on either rank selection or initialization in isolation—but also unifies both aspects within a single framework, enabling more effective and efficient adaptation. Extensive experiments across various architectures and modalities show that GoRA consistently outperforms existing LoRA-based methods while preserving the efficiency of vanilla LoRA. For example, when fine-tuning Llama3.1-8B-Base for mathematical reasoning, GoRA achieves a 5.13-point improvement over standard LoRA and even outperforms full fine-tuning by 2.05 points under high-rank settings.

1 Introduction

Open-source pre-trained large language models (LLMs) such as the Llama series [37, 9] have demonstrated exceptional capabilities. Through supervised fine-tuning, these models can be adapted to various downstream tasks such as code generation [35] and mathematical problem solving [44]. However, when the model has a parameter size ϕ and uses FP16/BF16 mixed-precision training strategy [29, 21] with the Adam optimizer [22], the parameters and gradients require 4ϕ bytes of memory, while the optimizer states require 12ϕ bytes. Thus, the minimum memory usage, excluding activations, reaches 16ϕ bytes. Such high memory demands limit the training of large language models under constrained resources. To reduce memory usage, Low-Rank Adaptation (LoRA) [18] decomposes the weight matrix $W \in \mathbb{R}^{m\times n}$ into $W = W_0 + \Delta W = W_0 + sAB$, where s is a scaling factor, and $A \in \mathbb{R}^{m\times r}, B \in \mathbb{R}^{r\times n}, r \ll \min(m, n)$, as shown in Figure 1(a). LoRA only updates the low-rank weights A and B, keeping W_0 unchanged, thereby significantly reducing the memory footprint of optimizer states. Although LoRA performs well on simple tasks when applied to pre-trained large language models, its performance on more challenging tasks such as mathematical reasoning and code generation still lags behind full fine-tuning [2, 11].

One of the critical factors in LoRA is its rank. Kalajdzievski et al. [20] demonstrate that increasing the rank of LoRA can significantly improve performance when paired with an appropriate scaling

*Corresponding author: chenlei@iim.ac.cn; †: Equal Contribution

Preprint. Under review.

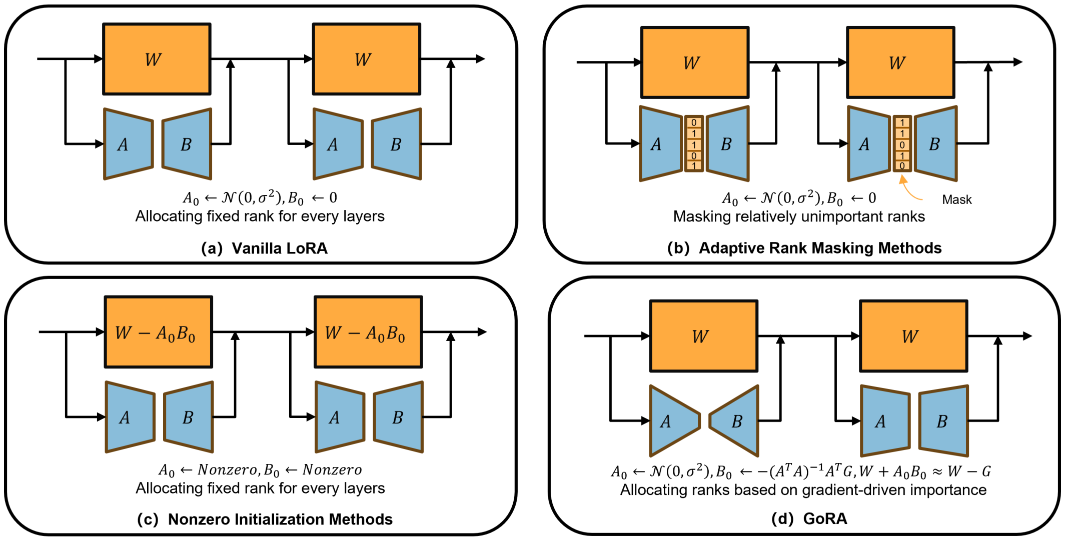
W
A
B
$A_0 \leftarrow \mathcal{N}(0,\sigma^2), B_0 \leftarrow 0$
Allocating fixed rank for every layers
(a) Vanilla LoRA
Mask
Masking relatively unimportant ranks
(b) Adaptive Rank Masking Methods
$W - A_0B_0$
$A_0 \leftarrow Nonzero, B_0 \leftarrow Nonzero$
(c) Nonzero Initialization Methods
$A_0 \leftarrow \mathcal{N}(0,\sigma^2), B_0 \leftarrow -(A^TA)^{-1}A^TG, W + A_0B_0 \approx W - G$
Allocating ranks based on gradient-driven importance
(d) GoRA

Figure 1: Illustration of (a) LoRA; (b) LoRA variants utilizing adaptive rank masking strategies; (c) LoRA variants employing nonzero initialization strategies; and (d) GoRA, which introduces adaptively leveraging weight W's gradient to allocate the adapter rank and initialize matrix B. A_0 and B_0 denote the initial value of matrix A and matrix B.

factor. However, a direct increase in rank leads to a substantial rise in memory requirement overhead, thus imposing constraints on rank selection. To address this, several studies [24, 34] propose to ensemble multiple low-rank subspaces, allowing for rank increases without proportionally increasing the number of trainable parameters. Nevertheless, these approaches often come at the expense of usability due to their intrusion into architecture or training processes. Another promising line of research explores adaptively assigning ranks to pre-trained weights based on importance. For example, AdaLoRA [48] adaptively adjusts ranks by quantifying the importance of each rank during training and masking less significant ones, as illustrated in Figure 1(b). However, this masking mechanism necessitates a larger parameter space (*e.g,* 1.5 times), increasing the number of trainable parameters and limiting the upper bound of rank. Consequently, as demonstrated in Section 5.4, adaptively allocating ranks without significantly increasing the training cost remains an open challenge.

Another vital factor in LoRA is its initialization strategy. LoRA initializes A_0 using a normal distribution (In PEFT library, A_0 is initialized with Kaiming distribution [15]) and B_0 with zeros. This initialization method ensures that the weights $W + A_0B_0$ remain unchanged at the beginning of training. Besides, zero initialization is not the only option: When A_0B_0 is nonzero, manipulating the pre-trained weight by subtracting A_0B_0 from W also ensures stability. Existing nonzero initialization methods can be categorized into experience-driven and data-driven methods. In experience-driven methods, PiSSA [27] and MiLoRA [39] employ decomposition techniques such as Singular Value Decomposition (SVD) to capture specific features of pre-trained weights. However, these methods are inherently task-agnostic, which limits their generalizability across diverse tasks. In contrast, data-driven methods incorporate task information. For example, LoRA-GA [40] uses the singular features of gradients to initialize LoRA matrices, minimizing the difference between LoRA and full fine-tuning. However, as illustrated in Figure 1(c) and Section 2.2, existing nonzero methods require manipulating the pre-trained weights, resulting in a training-inference gap. Thus, designing a nonzero initialization method without manipulating pre-trained weights remains an open problem.

Given the challenges of adaptive rank allocation and nonzero initialization, we turn to gradients of pre-trained weights, which are crucial for assessing the importance of pre-trained weights and deeply related to LoRA adapters' optimization processes [13, 50]. As shown in Figure 1(d), we propose GoRA. Specifically, before training, we compute gradients of pre-trained weights on a subset of training samples, using these gradients to assess the importance of each pre-trained weight. Given a reference rank, we calculate a trainable parameter budget. Based on the normalized importance and the trainable parameter budget, we allocate a new trainable parameter count and corresponding rank for each low-rank adapter, achieving adaptive rank allocation without significantly increasing the trainable parameter count compared to LoRA and allowing for higher rank allocation upper bounds, as shown in Table 5. In GoRA's initialization, we maintain LoRA's initialization for A, while B is initialized using

$-(A^TA)^{-1}A^TG$, where G is the gradient of pre-trained weight W. This initialization ensures that the computation result of the low-rank adapter $-A(A^TA)^{-1}A^TG \approx -G$ compresses the gradient optimally, setting a solid foundation for further optimization without manipulating the pre-trained weight. Our key contributions are summarized as follows:

1. We conduct an in-depth investigation into LoRA's rank allocation and initialization strategy, uncovering the limitations of existing works. We propose GoRA, which achieves adaptive rank allocation and initialization without compromising the usability and efficiency.
2. We use the gradients of weights to assess their importance and allocate ranks. We then initialize low-rank weights using the pseudo-inverse compressed gradients, enhancing performance while ensuring training stability.
3. We conduct extensive experiments, demonstrating that GoRA consistently outperforms low-rank baselines and even rivals full fine-tuning in certain settings. For example, on the Llama3.1-8B-Base model fine-tuned for mathematical reasoning, GoRA achieves a 5.13-point improvement over LoRA and even surpasses full fine-tuning high-rank configurations.

2 Related Works

2.1 Rank of LoRA

The choice of rank is crucial for LoRA, with higher ranks consistently yielding better outcomes [20]. However, increasing the rank raises the number of trainable parameters and corresponding memory usage overhead, making it challenging to train with sufficient ranks on limited hardware resources. Previous works [28, 24] attempt to continuously merge and reinitialize low-rank weights during training to stack the overall rank. However, these methods often require resetting the states of the optimizer and learning rate scheduler during reinitialization to ensure that updates take place in distinct low-rank subspaces and ensure training stability, significantly increasing overall training complexity and making the training process unstable. MeLoRA [34] proposes aggregating multiple mini low-rank adapters diagonally to increase the overall rank. Nevertheless, this approach requires modifying the structure of LoRA, limiting its usability.

At the same time, the significances of weights during training demonstrate heterogeneity, and an intuitive proposition is to assign larger ranks to relatively more important weights. Previous works [48, 19] attempted to dynamically mask less important ranks during training to achieve adaptive rank adjusting. However, these methods require allocating larger matrices for low-rank adapters to reserve space for masked ranks, leading to an increase in the number of trainable parameters, which compromises their operational efficacy and establishes limitations on the upper threshold of rank. IncreLoRA [46] introduces an approach that begins with a single rank for each low-rank adapter and incrementally increases the rank during training. This method effectively addresses the challenge of large initial matrices. Nevertheless, this approach demonstrates suboptimal compatibility with distributed training architectures, notably FSDP[51] and ZeRO[33], which constitute essential infrastructural components for the effective training of large-scale models.

2.2 Initialization of LoRA

Parameter initialization represents a fundamental paradigm in deep learning methodologies. Well-established initialization protocols, such as the strategy proposed by Xavier Glorot and Yoshua Bengio. [12], facilitate the convergent training trajectories of deep neural networks. Similarly, appropriate initialization strategies constitute a critical determinant for LoRA. Beyond zero initialization used by vanilla LoRA, some studies have explored different initialization strategies: PiSSA [27] performs SVD on pre-trained weights and uses the most important singular features to initialize low-rank weights; MiLoRA [39], in contrast to PiSSA, uses the least important singular features to initialize low-rank weights; similarly, OLoRA [3] uses QR decomposition of pre-trained weights to initialize low-rank weights; EVA [31] uses singular features of activations to initialize low-rank weights; and LoRA-GA [40] uses singular features of gradients to initialize low-rank weights. These methods can improve LoRA's performance to some extent.

Nevertheless, owing to the inherent non-zero initialization characteristics of these methodologies, they require subtracting the LoRA initialization results from the pre-trained weights to ensure correct forward and backward propagation during the initial phases of the training regimen, consequently creating a gap between training and inference. Recomputing the initialization result of these methods during inference is not feasible in cases involving randomness [27] or requiring original training data [40, 31]. And the initialization process requires significant time for methods such as MiLoRA [39]. The most straightforward solution is to save not only the low-rank weights but also the manipulated pre-trained weights, but this sacrifices one of LoRA's significant advantages, namely minimal checkpoint storage [10]. Another approach is to save the initialized LoRA weights and use block matrix multiplication to eliminate the gap, but this reduces usability.

3 Method

In this section, we will reinterpret LoRA adapters from the perspective of gradient compressors and introduce GoRA's gradient-guided adaptive rank allocation and initialization strategy.

3.1 View LoRA adapters as Gradient Compressors

The core idea of LoRA is to fine-tune a model by leveraging the intrinsic low-rank property of the update of a weight matrix $W \in \mathbb{R}^{m\times n}$ during training. Specifically, a pair of low-rank matrices $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{r\times n}$ are initialized alongside W. During training, W remains frozen, while the model is updated by training the low-rank matrices A and B, thereby reducing memory usage during training. For any training step t, the update to W is given by (1), where α is a tunable hyperparameter that ensures the scale of the LoRA computation depends only on α and is independent of the rank r:

$$W_t = W_0 + \Delta W = W_0 + \frac{\alpha}{r} A_t B_t. \tag{1}$$

Specifically, given the training loss L, the gradient of the weight matrix W can be computed as $\frac{\partial L}{\partial W}$. Using the chain rule, the gradients of A and B are $\frac{\partial L}{\partial W} B_t^T$ and $A_t^T \frac{\partial L}{\partial W}$, respectively. Given a learning rate η, the updates to the weight are as shown in (2)-(3):

$$\Delta B = -\eta \frac{\alpha}{r} \sum_{t=1}^{T} A_{t-1}^T \frac{\partial L_t}{\partial W_t}, \quad \Delta A = -\eta \frac{\alpha}{r} \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_t} B_{t-1}^T, \tag{2}$$

$$\Delta W = \frac{\alpha}{r} A_t B_t - \frac{\alpha}{r} A_0 B_0 = \frac{\alpha}{r}((A_0 + \Delta A)(\Delta B) - A_0 B_0) = \frac{\alpha}{r}(\Delta A \Delta B + A_0 \Delta B). \tag{3}$$

Experimental results from LoRA-FA [47] have shown that freezing the randomly initialized matrix A and only training matrix B can achieve performance close to that of LoRA. When matrix A is frozen ($\Delta A = 0$), the weight update is given by (4). One can observe that matrix B accumulates the gradients compressed by A^T during training, and when multiplied by A, the compressed gradients are up-projected. Thus, the training process of LoRA-FA can be viewed as a process of gradient accumulation and compression, with the compression matrix being the randomly initialized A.

$$\Delta W = \frac{\alpha}{r} A_0 \Delta B = -\eta \frac{\alpha}{r} \Sigma_{t=0}^{T} A_0 A_0^T \frac{\partial L_t}{\partial W_t}. \tag{4}$$

The update form of LoRA-FA provides significant inspiration. We hypothesize that vanilla LoRA has similar properties, i.e., LoRA adapters act as gradient compressors. Based on this hypothesis, we can allocate larger ranks to weights whose gradients and weights themselves contain more low-rank information and initialize LoRA parameters using compressed gradients.

3.2 GoRA's Adaptive Rank Allocation Strategy

Based on the hypothesis that LoRA adapters function similarly to gradient compressors, our adaptive rank allocation strategy aims to: (1) allocate rank based on weight importance derived from n-step accumulated gradients $G = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial L_i}{\partial W}$; (2) complete rank allocation before training to avoid dynamic shape changes; (3) maintain a similar number of trainable parameters as LoRA (within 10%); and (4) preserve structural compatibility with LoRA for easy integration.

To evaluate the importance of weights, we first consider the nuclear norm of the gradient, which aggregates all singular values σ of a matrix and is often used to measure low-rank properties [40].

However, as shown in Table 6, this metric does not effectively capture weight importance in practice. Instead, we adopt a sensitivity-based importance metric commonly used in model pruning [49]. Specifically, we define the importance of a weight matrix W as:

$$\mathrm{I}(W) = \mathrm{avg}(|W \odot G|), \tag{5}$$

where the operator $\odot$ denotes element-wise multiplication and $\mathrm{avg}(\cdot)$ computes the average value to yield a scalar importance score. After computing the importance scores for all target pre-trained weight matrices, we form an importance set $\{\mathrm{I}(W_i)\}_{i=1}^{N}$. To facilitate adaptive rank allocation, we normalize the importance set to compute an advantage A_i for i-th pre-trained weight W_i:

$$A_i = \frac{\mathrm{I}(W_i)}{\Sigma_{i=1}^{N}\mathrm{I}(W_i)}. \tag{6}$$

With the normalized advantages computed, we next determine the total trainable parameter budget B for the model. Given a reference rank r_{ref}, the budget for a single weight matrix $W_i \in \mathbb{R}^{m\times n}$ is estimated as:

$$B_i = (\sqrt{m+n}) \times r_{\text{ref}}, \tag{7}$$

reflecting the expected parameter cost under LoRA. Summing over all matrices, the total budget becomes $B = \Sigma_{i=1}^{N} B_i$. Using this budget and the advantage A_i, we allocate the adapter rank r_i and its trainable parameter count P_i for each weight matrix as:

$$r_i = \left[\frac{P_i}{\sqrt{m+n}}\right] = \left[\frac{B * A_i}{\sqrt{m+n}}\right], \quad \text{s.t.} r_{\text{min}} \le r_i \le r_{\text{max}}, \tag{8}$$

where the operator $[\cdot]$ denotes rounding to the nearest integer, and $r_{\text{min}}, r_{\text{max}}$ are hyper-parameters defining the allowable rank range. This formulation ensures that the total number of trainable parameters $P = \Sigma_{i=1}^{N} P_i$ closely matches that of standard LoRA with rank r_{ref}.

In summary, before training begins, we compute the n-step accumulated gradients for all target weights. These gradients are then used to estimate the importance of each weight matrix, based on which we perform adaptive rank allocation in GoRA — achieving all four objectives outlined earlier.

3.3 GoRA's Adaptive Initialization Strategy

Once ranks are allocated for each layer, it is crucial to properly initialize the low-rank matrices. The compression form in (4) is suboptimal when A is randomly initialized and fixed; to achieve better alignment with the gradient dynamics, we must initialize the B matrix such that the computation of the low-rank adapter at the start of training closely approximates the n-step accumulated gradient G. This optimal initialization can be derived using the Moore-Penrose inverse of A:

$$B = -(A^TA)^{-1}A^TG, \quad AB = -A(A^TA)^{-1}A^TG. \tag{9}$$

As shown in (9), initializing B as $-(A^TA)^{-1}A^TG$ ensures that AB provides the best low-rank approximation of G given a fixed A, **with proof provided in Appendix A.1**.

However, due to the properties of pseudo-inverse computation, the scale of AB does not exactly match that of G. Assuming both $G \in \mathbb{R}^{m\times n}$ and $A \in \mathbb{R}^{m\times r}$ follow distributions with mean 0 and variance 1, the expected Frobenius norm of G, $\mathbf{E}[||G||_F]$, is $\sqrt{mn}$, while that of AB, $\mathbf{E}[||AB||_F]$, is $\sqrt{rn}$, **as detailed in Appendix A.2**.

To ensure that the initial computation of the low-rank adapter approximates a single step of stochastic gradient descent with a tunable step size γ, we introduce a scaling factor ξ for B:

$$\frac{\alpha}{r}A(\xi B) \approx \xi\frac{\alpha}{r}\sqrt{\frac{r}{m}}G \approx -\gamma G. \tag{10}$$

Thus, to make GoRA's initialization equivalent to one step of gradient descent, ξ should be set to $\frac{\gamma\cdot\sqrt{rm}}{\alpha}$. Inspired by RSLoRA, and to better utilize the larger ranks obtained through dynamic allocation, we modify the forward computation formula to $W_t = W_0 + \Delta W = W_0 + \frac{\alpha}{\sqrt{r}}A_tB_t$, which adjusts ξ to $\frac{\gamma\cdot\sqrt{m}}{\alpha}$. Setting γ to a relatively large value further improves performance and yields optimal results. The full algorithm is summarized in Algorithm 1.

Algorithm 1 GoRA Rank Allocation and Initialization

```
Input: Model $f(\cdot)$ with $L$ layers, parameters $W$, gradient accumulation steps $N$, loss function $\mathcal{L}$,
    scale factor $\gamma$, Trainable parameter budget $B$
Output: Initialized low-rank matrices $A$, $B$
 1: for $l = 1$ to $L$ do
 2:     $G_l^{avg} \leftarrow 0$                              ▷ Initialize gradients buffer in CPU memory.
 3: for $i = 1$ to $N$ do          ▷ This operation do not require extra memory as shown in Table 8.
 4:     Randomly sampled mini-batch $B_i = \{x, y\}$
 5:     $\hat{y} \leftarrow f(x, W)$
 6:     $\ell \leftarrow \mathcal{L}(y, \hat{y})$
 7:     for $l = 1$ to $L$ do
 8:         $G_l^{avg} \leftarrow G_l^{avg} + \frac{1}{n}\frac{\partial \ell}{\partial W_l}$     ▷ Compute gradients and offload to cpu without optimizer step.
 9: for $l = 1$ to $L$ do
10:     Compute importance $I(W_l) \leftarrow \text{avg}(|W_l * G_l^{avg}|)$
11: for $l = 1$ to $L$ do
12:     Compute advantage $A_l \leftarrow \frac{I(W_l)}{\sum_{l=1}^{L} I(W_l)}$
13: for $l = 1$ to $L$ do
14:     $m, n \leftarrow \text{size}(W_l)$
15:     $r_l \leftarrow \text{clip}(\text{round}(\frac{B \cdot A_l}{\sqrt{m+n}}), r_{\text{min}}, r_{\text{max}})$        ▷ Clip by $r_{min}$ and $r_{max}$ to avoid extremum.
16:     $A_l \leftarrow \text{kaiming\_uniform}(m, r)$
17:     $B_l \leftarrow -(A_l^T A_l)^{-1} A_l^T G_l^{avg}$
18:     $\xi = \frac{\gamma \cdot \sqrt{m}}{\alpha}$
19:     $B_l = \xi \cdot B_l$
20: Return $A$, $B$
```

4 Experiments

We conducted comprehensive experiments comparing GoRA with baseline methods on natural language understanding (Section 4.1) , generation tasks (Section 4.2) and image classification tasks (Section 4.3). For understanding tasks, we trained T5-Base [32] on five tasks of GLUE [38] (MNLI, SST-2, CoLA, QNLI, MRPC) and reported accuracy on corresponding validation sets. For generation tasks, we fine-tuned Llama-3.1-8B-Base [9] and Llama-2-7B-Base [37] on chat, mathematics, and coding datasets, evaluating test performance on MTBench [52], GSM8K [7], and HumanEval [4]. For image classification tasks, we fine-tuned CLIP-ViT-B/16 on seven datasets including Stanford-Cars [23], DTD [6], EuroSAT [16], GT-SRB [17], RESISC45 [5], SUN397 [42] and SVHN [30] and reported test accuracy. All experiments used single-epoch training with three seeds, reporting mean values with standard deviations. Unless specified otherwise, we set the LoRA rank or GoRA's reference rank r_{ref} to 8. **The hyper-parameters of GoRA are detailed in Appendix B.3**.

4.1 Experimental Results on Natural Language Understanding Tasks

Table 1: Performance of fine-tuning T5-Base on 5 sub-tasks of the GLUE benchmark. **Bold** and underline indicate the highest and second-highest scores of low-rank methods with $r = 8$ or $r_{ref} = 8$.

Method	MNLI	SST-2	CoLA	QNLI	MRPC	Average
Full	86.33±0.00	94.75±0.21	80.70±0.24	93.19±0.22	84.56±0.73	87.91
LoRA [18]	85.30±0.04	94.04±0.11	69.35±0.05	92.96±0.09	68.38±0.01	82.08
Convergence Optimization Methods for LoRA						
RSLoRA [20]	85.73±0.10	94.19±0.23	72.32±1.12	93.12±0.09	52.86±2.27	79.64
DoRA [25]	85.67±0.09	94.04±0.53	72.04±0.94	93.04±0.06	68.08±0.51	82.57
LoRA+ [14]	85.81±0.09	93.85±0.24	77.53±0.20	93.14±0.03	74.43±1.39	84.95
Initialization Optimization Methods for LoRA						
PiSSA [27]	85.75±0.07	94.07±0.06	74.27±0.39	93.15±0.14	76.31±0.51	84.71
LoRA-GA [40]	85.70±0.09	94.11±0.18	**80.57±0.20**	93.18±0.06	85.29±0.24	87.77
Adaptive Methods for LoRA						
AdaLoRA [48]	85.45±0.11	93.69±0.20	69.16±0.24	91.66±0.05	68.14±0.28	81.62
GoRA	**85.91±0.02**	**94.68±0.43**	79.86±0.35	**93.27±0.08**	**86.10±0.20**	**87.96**

Settings: We adopted baseline performances reported by LoRA-GA [40], maintaining their experimental parameters for fair comparison: Adam[22] optimizer ($\beta_1 = 0.9, \beta_2 = 0.999$, weight decay $= 0$), batch size 32, cosine decay learning rate with 0.03 warmup ratio. We trained all linear layers except the language head using peak learning rate 1e-4, maximum sequence length 128, and FP32 precision.

Results: Table 1 compares GoRA against multiple baselines across five GLUE benchmark tasks. GoRA achieved superior performance on four datasets (MNLI, SST-2, QNLI, and MRPC), demonstrating exceptional adaptability and generalization. While slightly underperforming LoRA-GA on CoLA by just 0.71 percentage points, GoRA's average score (87.96) surpassed all baselines and even exceeded full fine-tuning (87.91). This confirms GoRA's ability to maximize model potential while maintaining parameter efficiency. Notably, GoRA showed particularly strong performance on MRPC and QNLI, highlighting its effectiveness in small-sample learning and sentence-pair tasks.

4.2 Experimental Results on Natural Language Generation Tasks

Settings: We trained mathematical, coding, and dialogue capabilities using 100K MetamathQA [45], 100K Code-FeedBack [53] (code-only labels), and 52K WizardLM [43] subsets, respectively. For experiments on Llama-3.1-8B-base, training used AdamW [26] ($\beta_1 = 0.9, \beta_2 = 0.999$, weight decay $= 5e - 4$) with batch size 64, cosine decay learning rate (warmup ratio=0.03, decay ratio=0.1), and BF16 mixed precision. For all methods including GoRA, we trained attention modules' linear components with peak learning rate 5e-5 (5e-4 for AdaLoRA). Evaluation metrics: mathematics—regex-extracted accuracy; coding—PASS@1; dialogue—average scores (0-10) from GPT-4o [1], Gemini-1.5-Pro [36], and Llama-3.1-70B-Instruct [9] using prompts from [52]. For experiments on Llama-2-7B-Base, we adopted baseline results from LoRA-GA [40], and we maintained the same training and evaluation settings. **Further details are provieded in Appendix B.4**.

Table 2: Performance of fine-tuning Llama-3.1-8B-Base.

Method	**MTBench**	**GSM8k**	**HumanEval**
Full	5.88±0.23	73.69±0.28	51.63±1.27
LoRA [18]	6.15±0.02	67.78±1.25	43.09±0.35
RSLoRA [20]	6.18±0.09	68.36±0.74	45.78±2.80
DoRA [25]	6.24±0.12	69.17±1.00	43.70±1.54
LoRA+ [14]	**6.35±0.10**	71.29±0.93	44.51±2.11
OLoRA [3]	6.13±0.04	68.54±0.42	43.29±2.44
PiSSA [27]	6.08±0.09	68.56±1.03	44.10±1.54
LoRA-GA [40]	5.99±0.06	71.39±0.90	43.29±0.61
AdaLoRA [48]	6.19±0.16	70.63±0.77	41.46±3.66
GoRA	6.34±0.04	**72.91±0.76**	**48.98±2.14**
GoRA$_{r_{ref}=32}$	6.21±0.10	75.59±1.04	51.22±1.83
GoRA$_{r_{ref}=128}$	5.82±0.31	75.74±0.40	52.03±1.41

Table 3: Performance of fine-tuning Llama-2-7B-Base.

Method	**MTBench**	**GSM8k**	**HumanEval**
Full	5.30 ± 0.11	59.36 ± 0.85	35.31 ± 2.13
LoRA [18]	5.61 ± 0.10	42.08 ± 0.04	14.76 ± 0.17
RSLoRA [20]	5.25 ± 0.03	45.62 ± 0.10	16.01 ± 0.79
DoRA [25]	$\mathbf{5.97} \pm \mathbf{0.02}$	53.07 ± 0.75	19.75 ± 0.41
LoRA+ [14]	5.71 ± 0.08	52.11 ± 0.62	18.17 ± 0.52
OLoRA [3]	5.30 ± 0.04	43.29 ± 0.83	17.22 ± 0.12
PiSSA [27]	5.30 ± 0.02	44.54 ± 0.27	16.02 ± 0.17
LoRA-GA [40]	5.95 ± 0.16	53.60 ± 0.30	19.81 ± 1.46
AdaLoRA [48]	5.57 ± 0.05	50.72 ± 1.39	17.80 ± 0.44
GoRA	5.61 ± 0.12	$\mathbf{54.04} \pm \mathbf{0.22}$	$\mathbf{24.80} \pm \mathbf{1.04}$
GoRA$_{r_{ref}=32}$	5.75 ± 0.06	56.18 ± 0.10	26.83 ± 2.84
GoRA$_{r_{ref}=128}$	6.05 ± 0.04	56.58 ± 0.12	27.85 ± 0.58

Results: Table 2 and Table 3 show the performance of GoRA and baseline methods on fine-tuned Llama3.1-8B-Base and Llama2-7B-Base. Specifically, GoRA demonstrated exceptional performance on the more challenging HumanEval and GSM8K benchmarks, substantially surpassing all baseline methods. For Llama3.1-8B-Base, on the GSM8K dataset, GoRA scored 72.91, outperforming LoRA-GA's 71.39 by 1.52 points; on the HumanEval dataset, GoRA achieved 48.98, surpassing RSLoRA's 45.78 by 3.20 points. On MTBench, GoRA slightly underperforms in terms of overall effectiveness, scoring 6.34—just 0.01 points lower than LoRA+'s 6.35. Notably, GoRA performed well across different rank allocation settings. For example, GoRA$_{r_{ref}=128}$ achieved 75.74 and 52.03 on the GSM8K and HumanEval, respectively, surpassing full fine-tuning's 73.69 and 51.63. Even the $r_{ref} = 32$ configuration of GoRA, while slightly underperforming $r_{ref} = 128$, still outperformed full fine-tuning on GSM8K. For Llama2-7B-Base, GoRA demonstrated similar superior results compared to baseline methods. Especially, GoRA outperformed LoRA-GA by 4.99

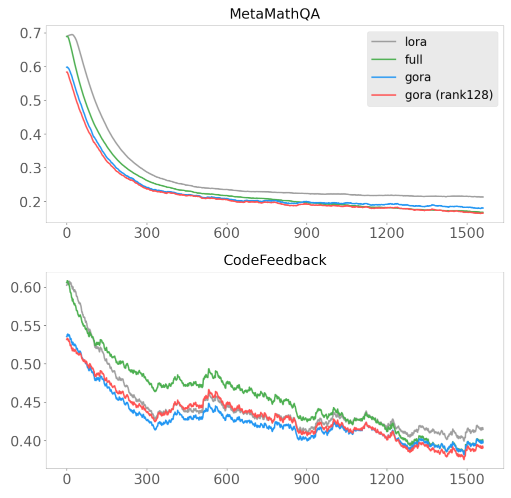

Figure 2: The training loss curves of full fine-tuning, LoRA, GoRA and GoRA$_{r_0=128}$ on Llama-3.1-8B-Base. **GoRA demonstrates lower start loss and faster convergence speed.**

and 0.44 on HumanEval and GSM8K, respectively. These results validate the effectiveness of GoRA across different LLMs and settings. The training loss curves of GoRA are depicted in Figure 2.

Table 4: Performance of fine-tuning CLIP-VIP-B/16 on 7 image classification tasks.

Method	**Cars**	**DTD**	**EuroSAT**	**GTSRB**	**RESISC45**	**SUN397**	**SVHN**	**Average**
Zero-shot	63.75	44.39	42.22	35.22	56.46	62.56	15.53	45.73
Full	84.23±0.06	77.44±0.19	98.09±0.03	94.31±0.28	93.95±0.00	75.35±0.10	93.04±0.18	88.06
LoRA [18]	72.81±0.13	73.92±0.38	96.93±0.07	92.40±0.10	90.03±0.14	70.12±0.18	88.02±0.07	83.46
DoRA [25]	73.72±0.06	73.72±0.33	96.95±0.01	92.38±0.08	90.03±0.08	70.20±0.19	88.23±0.05	83.48
LoRA+ [14]	72.87±0.18	74.07±0.45	97.01±0.02	92.42±0.18	89.96±0.11	70.17±0.15	88.08±0.05	83.51
LoRA-Pro [41]	**85.87±0.08**	**78.64±0.85**	98.46±0.03	95.66±0.05	94.75±0.21	76.42±0.14	94.63±0.20	89.20
LoRA-GA [40]	85.18±0.41	77.50±0.12	98.05±0.27	95.28±0.10	94.43±0.19	75.44±0.06	93.68±0.35	88.51
GoRA	85.76±0.19	78.17±0.32	**98.77±0.35**	**96.66±0.36**	**95.16±0.26**	**76.46±0.08**	**95.32±0.13**	**89.47**

4.3 Experimental Results on Image Classification Tasks

Settings: We adopted baseline performances from LoRA-Pro [41], maintaining their experimental hyper-parameters for a fair comparison: Adam [22] optimizer ($\beta_1 = 0.9, \beta_2 = 0.999$, weight decay $= 0$), batch size 64, cosine decay learning rate with 0.03 warmup ratio. We trained all linear layers in the vision backend using a peak learning rate of 1e-4, and FP32 precision. The classifier is obtained using prompts such as "a photo of a {class}."

Results: As shown in Table 4, GoRA outperforms baseline methods across all seven image classification tasks. Specifically, GoRA outperforms full fine-tuning by a margin of 1.01; outperforms LoRA-GA by 0.96 and outperforms LoRA-Pro by 0.27. These results demonstrate that GoRA exhibits superior performance across different models and modalities.

5 Discussion

In this section, we present a comprehensive set of ablation studies to evaluate the effectiveness of GoRA's adaptive rank allocation and initialization strategy. Additionally, we discuss the impact of hyper-parameters introduced by GoRA, providing insights into their roles in shaping the model's performance.

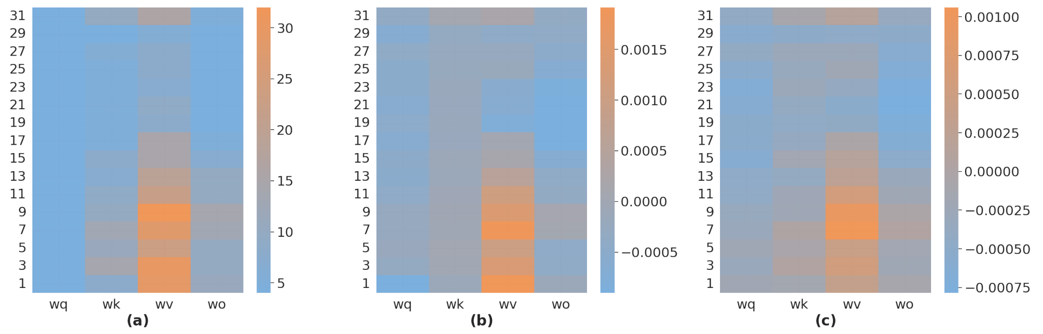

Figure 3: (a) Result rank distribution of fine-tuning Llama-3.1-8B-Base on the MetaMathQA-100K dataset using GoRA;(b) Difference values between GoRA and LoRA in directional updates of pre-trained weights after merging;(c) Difference values between GoRA and LoRA in magnitude updates of pre-trained weights after merging. Data points presented for every two layers.

5.1 The Effect of Rank Allocation Strategy.

The rank allocation strategy is a critical component influencing the performance of GoRA. As highlighted in Table 5, we conducted ablation studies to evaluate different rank allocation ranges. The results demonstrate that a broader rank allocation range consistently leads to superior performance. For instance, given $\gamma = 5e-2$, ($r_{min} = 4, r_{max} = 32$) achieved a score of 48.98 on HumanEval, significantly outperforming both the fixed rank allocation strategy ($r_{min} = 8, r_{max} = 8$) and the more conservative allocation strategy ($r_{min} = 6, r_{max} = 15$).

Figure 3 illustrates the rank distribution of ($r_{min} = 4, r_{max} = 32$). Notably, most ranks are allocated to the wv layers, while the wq layers receive the fewest rank allocations. This observation aligns with findings reported in prior work [18]. Moreover, weights with higher ranks receive larger updates after merging the low-rank matrices. These observations underscore the effectiveness of our rank allocation strategy.

Table 5: Ablation study on hyper-parameters. To maintain an approximately constant number of trainable parameters, the rank allocation upper bound was reduced as the lower bound was increased.

Method	r_{min}	r_{max}	γ	**GSM8k**	**HumanEval**
AdaLoRA	0	12	-	70.63±0.77	41.46±3.66
LoRA	8	8	0	67.78±1.25	43.09±0.35
GoRA	4	32	8e-2	**72.91±0.76**	46.54±1.54
GoRA	4	32	5e-2	72.88±0.99	**48.98±2.14**
GoRA	4	32	3e-2	72.71±1.22	45.93±1.27
GoRA	4	32	0	72.45±1.14	46.34±0.61
GoRA	0	∞	5e-2	72.83±0.80	46.13±3.36
GoRA	4	32	5e-2	72.88±0.99	**48.98±2.14**
GoRA	6	15	5e-2	72.25±0.27	45.85±3.18
GoRA	8	8	5e-2	72.10±1.12	44.75±3.97

5.2 The Effect of Initialization Strategy.

Table 5 also summarizes the results of ablation studies conducted with various scaling factors. Our experiments revealed that the choice of scaling factor γ has a substantial impact on performance. Notably, GoRA achieves the best performance on HumanEval with $\gamma = 5e - 2$, attaining a score of 48.98. Meanwhile, GoRA with $\gamma = 8e - 2$ slightly outperformed other configurations on the GSM8k, achieving a score of 72.91. Conversely, when $\gamma = 0$, GoRA exhibited the weakest performance on GSM8k, scoring 72.45. A carefully selected scaling factor ensures that the initial low-rank adapter computation closely approximates a gradient descent step, establishing a robust foundation for subsequent optimization. This is critical for training stability and superior performance.

5.3 The Effect of Different Importance Metrics.

Table 6 compares different importance metrics, including sensitivity of parameter to loss, gradient nuclear norm, and parameter-gradient product nuclear norm. The results show that parameter sensitivity consistently outperforms the other methods on GSM8k and HumanEval, particularly on the HumanEval dataset, where parameter sensitivity to loss achieved a score of 48.98, compared to 43.09 for gradient nuclear norm and 45.12 for parameter-gradient product nuclear norm.

Table 6: Ablation studies on different importance metrics, where $||\cdot||_*$ represents the nuclear norm.

Metric	**GSM8k**	**HumanEval**
avg($\|W * G\|$)	**72.88±0.99**	**48.98±2.14**
$\|\|W * G\|\|_*$	72.65±0.78	45.12±3.17
$\|\|G\|\|_*$	72.70±0.68	43.09±0.93

5.4 Computation and Memory Analysis

Table 7: Comparison of trainable parameters count. For GoRA, we set $r_{min} = 4$ and $r_{max} = 32$

Model	**Dataset**	**Target Modules**	**LoRA**	**AdaLoRA**	**GoRA**
T5-Base	SST-2	all-linear	3.24M	4.86M	3.05M
Llama3.1-8B-Base	MetamathQA	attention	6.82M	10.24M	7.00M
Llama2-7b-Base	MetamathQA	all-linear	19.99M	29.99M	20.18M
CLIP-ViT-B/16	Cars	vision_model	1.33M	2.03M	1.35M

To demonstrate the computation and memory efficiency of GoRA, we conduct experiments to measure both time and memory consumption. Specifically, we trained the Llama-3.1-8B-Base model on a 100K subset of MetaMathQA using a single RTX 4090 GPU, with a maximum sequence length of 512. As shown in Table 8, GoRA maintains similar training time (4 minutes for 64 gradient computing steps during initialization) and peak memory to LoRA, while the most widely used adaptive method AdaLoRA requires substantially more training time and memory. Furthermore, we benchmark trainable parameter counts across various models and tasks; the results are depicted in Table 7.

Table 8: Comparison of training time and peak GPU memory. Here, we consider the initialization process as a part of training process for GoRA.

Method	**Training Time**	**Peak Memory**
LoRA	5h50min	19.75GB
AdaLoRA	8h49min	19.93GB
GoRA	5h52min	19.75GB

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. “Gpt-4 technical report”. In: *arXiv preprint arXiv:2303.08774* (2023).

[2] Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, et al. “Lora learns less and forgets less”. In: *arXiv preprint arXiv:2405.09673* (2024).

[3] Kerim Büyükakyüz. “OLoRA: Orthonormal Low-Rank Adaptation of Large Language Models”. In: *arXiv preprint arXiv:2406.01775* (2024).

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[5] Gong Cheng, Junwei Han, and Xiaoqiang Lu. “Remote sensing image scene classification: Benchmark and state of the art”. In: *Proceedings of the IEEE* 105.10 (2017), pp. 1865–1883.

[6] Mircea Cimpoi, Subhransu Maji, Iasonas Kokkinos, Sammy Mohamed, and Andrea Vedaldi. “Describing textures in the wild”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 3606–3613.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).

[8] Tri Dao. “Flashattention-2: Faster attention with better parallelism and work partitioning”. In: *arXiv preprint arXiv:2307.08691* (2023).

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).

[10] Vlad Fomenko, Han Yu, Jongho Lee, Stanley Hsieh, and Weizhu Chen. “A Note on LoRA”. In: *arXiv preprint arXiv:2404.05086* (2024).

[11] Sreyan Ghosh, Chandra Kiran Reddy Evuru, Sonal Kumar, Deepali Aneja, Zeyu Jin, Ramani Duraiswami, Dinesh Manocha, et al. “A Closer Look at the Limitations of Instruction Tuning”. In: *arXiv preprint arXiv:2402.05119* (2024).

[12] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[13] Yongchang Hao, Yanshuai Cao, and Lili Mou. “Flora: Low-Rank Adapters Are Secretly Gradient Compressors”. In: *arXiv preprint arXiv:2402.03293* (2024).

[14] Soufiane Hayou, Nikhil Ghosh, and Bin Yu. “Lora+: Efficient low rank adaptation of large models”. In: *arXiv preprint arXiv:2402.12354* (2024).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[16] Patrick Helber, Benjamin Bischke, Andreas Dengel, and Damian Borth. “Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.7 (2019), pp. 2217–2226.

[17] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. “Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark”. In: *The 2013 international joint conference on neural networks (IJCNN)*. Ieee. 2013, pp. 1–8.

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. “Lora: Low-rank adaptation of large language models”. In: *arXiv preprint arXiv:2106.09685* (2021).

[19] Yahao Hu, Yifei Xie, Tianfeng Wang, Man Chen, and Zhisong Pan. “Structure-aware low-rank adaptation for parameter-efficient fine-tuning”. In: *Mathematics* 11.20 (2023), p. 4317.

[20] Damjan Kalajdzievski. “A rank stabilization scaling factor for fine-tuning with lora”. In: *arXiv preprint arXiv:2312.03732* (2023).

[21] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. "A study of BFLOAT16 for deep learning training". In: *arXiv preprint arXiv:1905.12322* (2019).

[22] Diederik P Kingma. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[23] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. "3d object representations for fine-grained categorization". In: *Proceedings of the IEEE international conference on computer vision workshops*. 2013, pp. 554–561.

[24] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. "Relora: High-rank training through low-rank updates". In: *The Twelfth International Conference on Learning Representations*. 2023.

[25] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. "Dora: Weight-decomposed low-rank adaptation". In: *arXiv preprint arXiv:2402.09353* (2024).

[26] I Loshchilov. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[27] Fanxu Meng, Zhaohui Wang, and Muhan Zhang. "Pissa: Principal singular values and singular vectors adaptation of large language models". In: *arXiv preprint arXiv:2404.02948* (2024).

[28] Xiangdi Meng, Damai Dai, Weiyao Luo, Zhe Yang, Shaoxiang Wu, Xiaochen Wang, Peiyi Wang, Qingxiu Dong, Liang Chen, and Zhifang Sui. "Periodiclora: Breaking the low-rank bottleneck in lora optimization". In: *arXiv preprint arXiv:2402.16141* (2024).

[29] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017).

[30] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Baolin Wu, Andrew Y Ng, et al. "Reading digits in natural images with unsupervised feature learning". In: *NIPS workshop on deep learning and unsupervised feature learning*. Vol. 2011. 2. Granada. 2011, p. 4.

[31] Fabian Paischer, Lukas Hauzenberger, Thomas Schmied, Benedikt Alkin, Marc Peter Deisenroth, and Sepp Hochreiter. "One initialization to rule them all: Fine-tuning via explained variance adaptation". In: *arXiv preprint arXiv:2410.07170* (2024).

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *Journal of machine learning research* 21.140 (2020), pp. 1–67.

[33] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "Zero: Memory optimizations toward training trillion parameter models". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–16.

[34] Pengjie Ren, Chengshun Shi, Shiguang Wu, Mengqi Zhang, Zhaochun Ren, Maarten Rijke, Zhumin Chen, and Jiahuan Pei. "MELoRA: mini-ensemble low-rank adapters for parameter-efficient fine-tuning". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2024, pp. 3052–3064.

[35] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. "Code llama: Open foundation models for code". In: *arXiv preprint arXiv:2308.12950* (2023).

[36] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context". In: *arXiv preprint arXiv:2403.05530* (2024).

[37] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288* (2023).

[38] Alex Wang. "Glue: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461* (2018).

[39] Hanqing Wang, Yixia Li, Shuo Wang, Guanhua Chen, and Yun Chen. "Milora: Harnessing minor singular components for parameter-efficient llm finetuning". In: *arXiv preprint arXiv:2406.09044* (2024).

[40] Shaowen Wang, Linxi Yu, and Jian Li. "Lora-ga: Low-rank adaptation with gradient approximation". In: *arXiv preprint arXiv:2407.05000* (2024).

[41] Zhengbo Wang, Jian Liang, Ran He, Zilei Wang, and Tieniu Tan. "LoRA-Pro: Are Low-Rank Adapters Properly Optimized?" In: *arXiv preprint arXiv:2407.18242* (2024).

[42] Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. "Sun database: Large-scale scene recognition from abbey to zoo". In: *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE. 2010, pp. 3485–3492.

[43] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. "WizardLM: Empowering large pre-trained language models to follow complex instructions". In: *The Twelfth International Conference on Learning Representations*. 2024.

[44] An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. "Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement". In: *arXiv preprint arXiv:2409.12122* (2024).

[45] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. "Metamath: Bootstrap your own mathematical questions for large language models". In: *arXiv preprint arXiv:2309.12284* (2023).

[46] Feiyu Zhang, Liangzhi Li, Junhao Chen, Zhouqiang Jiang, Bowen Wang, and Yiming Qian. "Increlora: Incremental parameter allocation method for parameter-efficient fine-tuning". In: *arXiv preprint arXiv:2308.12043* (2023).

[47] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. "Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning". In: *arXiv preprint arXiv:2308.03303* (2023).

[48] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. "AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning". In: *arXiv preprint arXiv:2303.10512* (2023).

[49] Qingru Zhang, Simiao Zuo, Chen Liang, Alexander Bukharin, Pengcheng He, Weizhu Chen, and Tuo Zhao. "Platon: Pruning large transformer models with upper confidence bound of weight importance". In: *International conference on machine learning*. PMLR. 2022, pp. 26809–26823.

[50] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. "Galore: Memory-efficient llm training by gradient low-rank projection". In: *arXiv preprint arXiv:2403.03507* (2024).

[51] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. "Pytorch fsdp: experiences on scaling fully sharded data parallel". In: *arXiv preprint arXiv:2304.11277* (2023).

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. "Judging llm-as-a-judge with mt-bench and chatbot arena". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 46595–46623.

[53] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. "Opencodeinterpreter: Integrating code generation with execution and refinement". In: *arXiv preprint arXiv:2402.14658* (2024).

A Proofs

A.1 Proof of optimal approximation of G given A.

Let G be an $m \times n$ matrix, and A be an $m \times r$ matrix where $r \ll \min(m, n)$. We aim to derive the projection formula that minimizes the Frobenius norm of the error $\|G - \hat{G}\|_F$, where $\hat{G}$ is the optimal approximation of G in the column space of A, denoted as $\text{Col}(A)$.

The best approximation $\hat{G}$ lies in $\text{Col}(A)$, so we can express $\hat{G}$ as:

$$\hat{G} = AB,$$

where B is an $r \times n$ matrix of coefficients to be determined. Our goal is to find B such that the error $\|G - \hat{G}\|_F$ is minimized.

The error matrix is given by:

$$E = G - \hat{G} = G - AB.$$

To minimize $\|E\|_F^2$, we take the derivative of $\|E\|_F^2$ with respect to B and set it to zero. Expanding $\|E\|_F^2$, we have:

$$\|E\|_F^2 = \text{Tr}\left((G - AB)^T (G - AB)\right),$$

where Tr represents the trace of a matrix.

Expanding this expression:

$$\|E\|_F^2 = \text{Tr}(G^T G) - 2\text{Tr}(B^T A^T G) + \text{Tr}(B^T A^T AB).$$

Taking the derivative with respect to B and setting it to zero:

$$-2A^T G + 2A^T AB = 0.$$

Simplifying:

$$A^T AB = A^T G.$$

Assuming $A^T A$ is invertible, we solve for B:

$$B = (A^T A)^{-1} A^T G.$$

Substituting B into $\hat{G} = AB$, we get:

$$\hat{G} = A(A^T A)^{-1} A^T G.$$

Thus, the best approximation $\hat{G}$ is:

$$\hat{G} = A(A^T A)^{-1} A^T G.$$

The matrix $\hat{G} = A(A^T A)^{-1} A^T G$ is the projection of G onto the column space of A, and it minimizes the Frobenius norm of the error $\|G - \hat{G}\|_F$.

A.2 Proof of Expectation of Frobenius Norm of AB.

Let A be a random Gaussian matrix of size $m \times r$, where each element of A is sampled independently from $\mathcal{N}(0, 1)$. Let G be a random Gaussian matrix of size $m \times n$, where each element of G is also sampled independently from $\mathcal{N}(0, 1)$. Define:

$$B = (A^\top A)^{-1} A^\top G,$$

and consider the product:

$$AB = A(A^\top A)^{-1} A^\top G.$$

The goal is to compute the expected Frobenius norm $\mathbb{E}[\|AB\|_F]$, where the Frobenius norm is defined as:

$$\|AB\|_F = \sqrt{\sum_{i,j} (AB)_{ij}^2}.$$

First, observe that AB can be rewritten as:

$$AB = A(A^\top A)^{-1}A^\top G.$$

Let $P = A(A^\top A)^{-1}A^\top$. Note that P is a projection matrix onto the column space of A, and thus P satisfies:

$$P^2 = P, \quad P^\top = P, \quad \text{and} \quad \text{rank}(P) = r.$$

Substituting P into the expression for AB, we have:

$$AB = PG.$$

The Frobenius norm of AB is given by:

$$\|AB\|_F^2 = \|PG\|_F^2 = \text{Tr}((PG)(PG)^\top).$$

Since $(PG)^\top = G^\top P$, this becomes:

$$\|AB\|_F^2 = \text{Tr}(PGG^\top P).$$

The matrix $GG^\top$ is a $m \times m$ random Wishart matrix. When G is a standard Gaussian matrix of size $m \times n$, the expected value of $GG^\top$ is:

$$\mathbb{E}[GG^\top] = n \cdot I_m,$$

where I_m is the $m \times m$ identity matrix. Substituting this result into the expression for $\|AB\|_F^2$, we get:

$$\mathbb{E}[\|AB\|_F^2] = \mathbb{E}[\text{Tr}(PGG^\top P)] = \text{Tr}(P\mathbb{E}[GG^\top]P).$$

Using $\mathbb{E}[GG^\top] = n \cdot I_m$, this simplifies to:

$$\mathbb{E}[\|AB\|_F^2] = \text{Tr}(P(n \cdot I_m)P) = n \cdot \text{Tr}(P^2).$$

Since $P^2 = P$, we have:

$$\mathbb{E}[\|AB\|_F^2] = n \cdot \text{Tr}(P).$$

The trace of P is equal to its rank, which is the dimension of the column space of A. Since A is a $m \times r$ matrix, we have:

$$\text{Tr}(P) = r.$$

Thus:

$$\mathbb{E}[\|AB\|_F^2] = n \cdot r.$$

Taking the square root, the expected Frobenius norm of AB is:

$$\mathbb{E}[\|AB\|_F] = \sqrt{n \cdot r}.$$

B Experimental Details

B.1 Baseline Methods

We compared GoRA with baseline methods to demonstrate the effectiveness of our approach:

a. **Full**: Trains all parameters in the target layers, resulting in the highest memory consumption.

b. **LoRA** [18]: Introduces low-rank adapters into the target layers, significantly reducing the number of trainable parameters.

c. **Convergence Optimization Methods for LoRA**
   - **RSLoRA** [20]: Modifies the scaling factor in LoRA from $\frac{\alpha}{r}$ to $\frac{\alpha}{\sqrt{r}}$, enabling better performance with higher-rank adapters and stabilizing the training processes.
   - **DoRA** [25]: Decomposes the weight updates of pre-trained weights into magnitude and direction components, and applies LoRA to update only the direction.
   - **LoRA+** [14]: Addresses the imbalance between matrices A and B in LoRA by assigning a relatively larger learning rate to matrix B than to matrix A.

d. **Initialization Optimization Methods for LoRA**
- **OLoRA** [3]: Initializes LoRA weights using the QR decomposition of the corresponding pre-trained weights.
- **PiSSA** [27]: Initializes LoRA weights based on the dominant singular vectors obtained from the SVD of pre-trained weights.
- **LoRA-GA** [40]: Initializes LoRA weights using significant singular vectors derived from the SVD of gradients of pre-trained weights.

e. **Adaptive Methods for LoRA**
- **AdaLoRA** [48]: Approximates the low-rank adapter structure using SVD, enabling dynamic rank allocation through singular value masking. It also introduces an orthogonal regularization term to the loss function to promote orthogonality among features in the low-rank adapter.

B.2 Implementation Details for Baseline Methods

Several baseline methods introduce tunable hyper-parameters compared with vanilla LoRA [18]. To ensure a fair comparison, we adopt the optimal settings reported in the original papers whenever possible. Specifically, for LoRA+ [14], we set the learning rate ratio of matrices A and B to 16. For LoRA-GA [40], we use the "stable" scaling method and manipulate the pre-trained weights during initialization. For AdaLoRA [48], the initial rank is set to 12, the final rank to 8, with $t_i = 150$ and $t_f = 900$. For PiSSA [27], the number of iterations for fast SVD is set to 64.

B.3 Implementation Details for GoRA

For all experiments with $r_0 = 8$, except for the model trained on MetaMathQA [45], we set the scaling factor γ to $5e-2$. For the model trained on MetaMathQA, γ is set to $8e-2$. For all experiments with $r_0 = 32$, the scaling factor is set to $1e-2$; and for $r_0 = 128$, we set the scaling factor to $5e-3$. This is because we observe that more gradient information is compressed by GoRA's initialization with higher rank even if γ can control the magnitude of the initialization results. To address the imbalance in GoRA's matrices A and B, we set the learning rate of matrix B to be 16 times that of matrix A Throughout the experiments, the r_{max} was empirically defined as $4 \times r_0$, and the r_{min} was set to $r_0/2$ (as this setting can maintain a comparable parameter count compared to LoRA), and the gradient accumulation step for GoRA's initialization was set to 64. In the ablation studies, we adhered to the same hyperparameter settings as in the main experiments, unless otherwise specified.

B.4 Hyper-parameters for Each Experiment

The hyper-parameters used in each experiment are summarized in Table 9. For experiments on T5-Base and Llama2-7B-Base, we adopt the settings from LoRA-GA [40]; for CLIP-ViT-B/16, we follow LoRA-Pro [41]. For experiments on Llama3.1-8B-Base, including both baseline methods and GoRA, we use one of the most commonly adopted hyperparameter configurations.

Table 9: Hyper-parameters used in experiments

Model	**LR**	**LR Decay**	**Warmup**	**Optimizer**	**Betas**	**Weight Decay**	**Batch Size**
T5-Base	1e-4	0	0.3	Adam	0.9, 0.999	0	32
Llama3.1-8B-Base	5e-5	0.1	0.3	AdamW	0.9, 0.999	5e-4	64
Llama2-7B-Base	2e-5	0	0.3	AdamW	0.9, 0.999	0	32
CLIP-ViT-B/16	1e-4	0	0.3	Adam	0.9, 0.999	0	64

B.5 Training Environments

For natural language understanding tasks reported in section 4.1, we conduct our experiments using the Huggingface Transformers framework for model and trainer implementation on a single accelerator which is comparable to RTX 4090 24GB. In contrast, for natural language generation tasks reported in section 4.2 and section 5, we utilize the DeepSpeed ZeRO2 [33] data parallel framework and FlashAttention-2 [8] mechanism, leveraging the power of 8 accelerators which are comparable to RTX 4090 24GB in a Slurm cluster. All codes of GoRA and baseline methods are implemented in PyTorch.

C Clarifications

C.1 Clarification on the training-inference gap introduced by previous initialization methods

This is due to their reliance on manipulating pre-trained weights. Specifically:

Manipulation of Pre-Trained Weights: These methods are required to manipulate the value of pre-trained weights during initialization, as $A_0B_0 \neq 0$ and $W_0 + A_0B_0 \neq W_0$. As a result, during inference, the term A_0B_0 must be recomputed in order to properly reconstruct the adapted weight matrix for effective model deployment, which is essential for correct model outputs.

The Inconvenience of Recalculating Initialization Results: During inference, it is often infeasible to recompute the initialization results for methods that either rely on randomness, such as PiSSA[27], or require access to training data, such as LoRA-GA [40] and EVA[31]. Furthermore, for approaches like MiLoRA [39], the initialization process itself can be computationally expensive and time-consuming.

Incompatibility Across Multiple Adapters: When multiple adapters are trained on different tasks using previous data-driven non-zero initialization methods, the pre-trained weights are manipulated inconsistently. As the result of A_0B_0 depends on the task. This makes it challenging to serve multiple adapters simultaneously, limiting flexibility in multi-task scenarios.

Saving manipulated pre-trained weights sacrifice one of the key adavantages of LoRA: While it is possible to merge the low-rank adapter weights into the pre-trained weights after training, saving the pre-trained weights post-merging sacrifices one of the key advantages of LoRA: minimal storage requirements (e.g., 10MB compared to 14GB). Other potential approaches to eliminate the training-inference gap and their limitations are discussed in Section 2.2.

C.2 Compare GoRA's initialization strategy with LoRA-GA

Both GoRA and LoRA-GA leverage gradients to initialize low-rank adapters, but there are several key differences:

1. **Motivation**:
   - LoRA-GA: Minimizes the difference of updates of weights between LoRA and full fine-tuning.
   - GoRA: Views LoRA adapters gradient compressors and optimizes the compression form.
2. **Scaling factor**:
   - LoRA-GA: Inspired by RSLoRA, its scaling aims to stabilize training.
   - GoRA: Scale the initialized product of adapters to any desired magnitude flexibly.
3. **Methodology**:
   - LoRA-GA: Initialize the weights using SVD decomposed gradients.
   - GoRA: Initialize the weights of B using gradients compressed by pseudo-inverse of random initialized A.

C.3 Further compare with previous related works

Build upon previous works, GoRA makes several unique contributions:

1. First data-driven initialization method without manipulating pre-trained weights.
2. Efficient rank allocation strategy without additional trainable parameters and training complexity.
3. Unified framework for gradient-driven rank allocation and initialization.

D Limitations And Future Works

In this study, we have demonstrated that GoRA outperforms baseline low-rank adaptation methods and achieves performance comparable to full fine-tuning. However, our evaluation has not yet extended to larger models and more extensive datasets. We hypothesize that for larger models, such as Llama-3.1-70B [9], GoRA could more effectively leverage the pre-trained knowledge inherent in these models. Additionally, while this research primarily focuses on language models and natural language processing tasks, there is potential to generalize GoRA to a broader range of model types and tasks such as visual language models and visual question answering.

Another limitation of this study is that the initialization of matrix A is not restricted to random initialization. Employing alternative methods, such as extracting distinguishing features from pre-trained weights to initialize matrix A, could potentially enhance performance, as it would combine the benefits of both experience-driven and data-driven initialization approaches. Furthermore, it is worth noting that GoRA demonstrates theoretical compatibility with other LoRA variants, such as DoRA [25]. These promising avenues remain to be explored in future research endeavors.