BI-SHARE LORA: ENHANCING THE PARAMETER EFFICIENCY OF LORA VIA INTRA-LAYER AND INTER-LAYER SHARING

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-Rank Adaptation (LoRA) is a widely adopted parameter-efficient fine-tuning method for large language models (LLMs) to adapt to downstream tasks. However, in scenarios where multiple LoRA models are deployed simultaneously, standard LoRA introduces substantial trainable parameters, resulting in significant memory overhead and inference latency, particularly when supporting thousands of downstream tasks on a single server. While existing methods reduce stored parameters via parameter sharing, they fail to capture both local and global information simultaneously. To address this issue, we propose Bi-Share LoRA, which integrates local parameters with intra-layer and inter-layer shared parameters to more effectively capture information at both local and global levels. By sharing parameters both within and across layers, our method significantly reduces the number of trainable parameters while preserving or improving model performance. Additionally, we set a local LoRA to capture local parameters, enabling more precise and fine-grained information extraction at the local level. The final implementation introduces three parallel sub-LoRAs and designs transformation techniques to adapt shared parameters of varying shapes, ensuring compatibility and efficient sharing. Experiments on the 7B, 8B, and 13B versions of Llama show that Bi-Share LoRA, with only 44.59% of the parameters of standard LoRA, outperforms LoRA by approximately 0.33% on commonsense reasoning and 2.08% on MMLU benchmarks.

1 INTRODUCTION

Large language models (LLMs), such as GPT-4o (Openai, 2023) and Claude-3 (Anthropic, 2024), have recently demonstrated remarkable generalization capabilities across a wide range of natural language tasks (Raiaan et al., 2024; Chang et al., 2024; Zhang et al., 2023). This enhanced performance is largely attributed to the rapid increase in model parameters, for example, GPT-3 (Brown et al., 2020) contains 175 billion parameters, while the largest version of Llama 3.1 (Touvron et al., 2023) features up to 405 billion parameters. However, the ever-increasing size of these models presents significant challenges for fine-tuning, as full parameter fine-tuning becomes computationally expensive and memory-intensive. To address this issue, Parameter-Efficient Fine-Tuning (PEFT) methods have been introduced, achieving performance comparable to full fine-tuning by adjusting only a small subset of parameters (Han et al., 2024) while keeping the majority of the model parameters frozen. Among these methods, LoRA (Hu et al., 2022) stands out by approximating parameter updates using the product of two low-rank matrices and has gained increasing popularity.

However, with the continued scaling up of models' parameters, fine-tuning LLMs with LoRA would introduce a considerable number of additional parameters, even when using a relatively low rank. For instance, fine-tuning the Llama 70B model with a LoRA rank of 64 introduces approximately 360 million parameters (1.4G of memory), which makes the training process more challenging, an issue that becomes more severe when multiple LoRA services are deployed simultaneously (Wang et al., 2024). For the deployment scenario, LoRA modules are often kept separate from the pre-trained parameters to facilitate multi-task inference services (Chen et al., 2024). Due to the multitude of downstream tasks, storing numerous LoRA weight backups consumes significant storage space. Additionally, during inference, the excessive quantity of LoRA parameters can occupy sub-

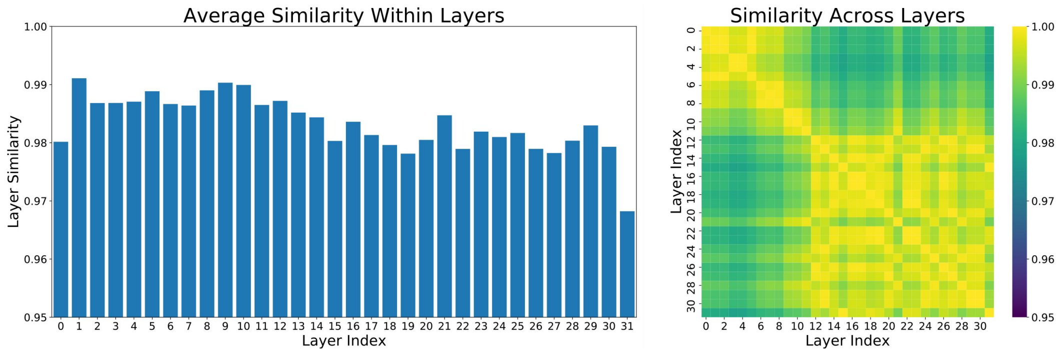

Figure 1: The entropy similarity (Lin et al., 2024) of LoRA parameters for each module within the same layer (left) and across different layers (right). It shows that different modules within the same layer exhibit high entropy similarity, and this high similarity is also present across different layers. This indicates that LoRA parameters have a significant degree of redundancy.

stantial memory, increasing latency when loading and switching between weights for various tasks. Consequently, there is a pressing need to further reduce the trainable parameters of LoRA.

Several existing methods target reducing LoRA's parameters through parameter-sharing strategies (Mao et al., 2024; Sun et al., 2022a). Among these methods, inter-layer parameter sharing has proven to be an effective technique for reducing redundancy across different layers. Several methods like VeRA (Kopiczko et al., 2024) and VB-LoRA (Li et al., 2024) reduce the memory and compute requirements by sharing parameters across layers, capturing global patterns that exist throughout the model. However, these methods often overlook the local information and a critical source of redundancy: the intra-layer redundancy present in the different submodules of the same layer (Lin et al., 2024). For example, in Transformer models, the attention heads and feed-forward networks in the same layer often process similar features, leading to redundant parameter usage. **Therefore, there is a need to design a new sharing technique that can capture both local and global features, while ensuring the shared parameters can be adapted to all modules with different shapes.**

In this paper, we first analyze LoRA parameters and identify a high degree of redundancy (Figure 1), and we conduct a preliminary study, as shown in Table 1, which demonstrates that sharing parameters within and between layers can achieve comparable performance with fewer parameters. **This indicates that we can eliminate high degree of redundancy in LoRA parameters by sharing them to capture global information.** Accordingly, we propose Bi-Share LoRA, a method that combines intra-layer and inter-layer parameter sharing. We decompose the LoRA matrices into three components: local parameters, which capture module-specific information, intra-layer shared parameters, which are shared within the same layer to capture local consistent features, and inter-layer shared parameters, which are shared across layers to capture global patterns. This enables Bi-Share LoRA to learn both local and global information efficiently. Additionally, to tackle the challenge of adapting shared parameters to all modules with different shapes, we present three shape transformation methods: Slice Sharing, Gate Transformation, and Kronecker Extension. To validate the effectiveness of Bi-Share LoRA, we conduct extensive experiments on the Llama model family across multiple commonsense reasoning and MMLU benchmarks. Our results demonstrate that Bi-Share LoRA achieves significant parameter savings of about 50% while maintaining or even improving the model's performance compared to standard LoRA and other existing methods. We also conduct experiments to analyze the rank value benefits and contributions of different configurations for local and shared weights.

In summary, our contributions are as follows:

- We propose Bi-Share LoRA, a unified sharing method that combines local parameters with intra-layer and inter-layer shared parameters to effectively capture both local and global information. This approach significantly reduces the number of trainable parameters while maintaining performance.
- We introduce three shape transformation techniques to handle varying parameter shapes, thus increasing the flexibility and effectiveness of parameter sharing.

- We conduct extensive experiments on multiple tasks, demonstrating the effectiveness of Bi-Share LoRA in reducing parameter redundancy and improving parameter efficiency.

2 Background and Motivation

2.1 Low-Rank Adaptation

LoRA fine-tuning is employed to recover performance with minimal parameter updates. For an LLM consisting of l layers, the weight matrix of each layer $\boldsymbol{W}$ is adjusted using an update matrix $\Delta\boldsymbol{W} \in \mathbb{R}^{m\times n}$. This matrix is factorized into two low-rank matrices, $\boldsymbol{A}$ and $\boldsymbol{B}$, where $\boldsymbol{A} \in \mathbb{R}^{r\times m}$ and $\boldsymbol{B} \in \mathbb{R}^{n\times r}$, with r being a hyperparameter shared by all layers. The effectiveness of fine-tuning is highly dependent on rank selection. In this approach, the original weight matrix $\boldsymbol{W}$ remains frozen, while only $\Delta\boldsymbol{W}$, represented by the product $\boldsymbol{AB}$, is updated. The forward computation can be expressed as:

$$f(\boldsymbol{x}) = (\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{A}\boldsymbol{x} \ . \tag{1}$$

Given that the rank r is typically much smaller than the dimension d, the computational cost is significantly reduced from d^2 to $2dr$. This optimization can reduce the trainable parameters during the learning process. Typically, the matrix $\boldsymbol{A}$ is initialized by a Gaussian distribution with a small standard deviation, and $\boldsymbol{B}$ is initialized as a zero matrix. Hence, at the beginning of fine-tuning, the model behaves identically to the pre-trained model.

2.2 Motivation

In large language models (LLMs), parameter redundancy is a common issue, especially in multi-task learning scenarios. Redundancy commonly occurs within the same transformer block where the attention layer and MLP layer have overlapping functions or learn similar patterns (Lin et al., 2024), and the parameters across different blocks where similar feature representations might be learned in multiple layers (Li et al., 2024). We also plot the parameter similarity in Figure 1, which shows the high similarity across different modules (details refer to Appendix A.2). Addressing both scenarios of parameter redundancy remains an open problem. We wonder ***whether sharing these superfluous parameters, both within a layer (intra-layer sharing) and across layers (inter-layer sharing), can significantly reduce the number of parameters without sacrificing performance?*** To this end, we conduct a simple experiment on intra-layer sharing, where all modules within the same transformer layer share the same LoRA parameters, and inter-layer sharing, where modules across different layers share the same LoRA fine-tuning parameters. From Table 1 (Individual), we find that the simple sharing may result in some performance degradation. We hypothesize that it may need module-specific parameters to learn the local features.

Table 1: Performance on instruction tuning with Alpaca 50K (Taori et al., 2023), evaluated with MMLU (Hendrycks et al., 2021) and GSM8K (Cobbe et al., 2021). **Boldface** indicates the best performance.

Method	r_{local}	r_{intra}	r_{inter}	# params	ratio	MMLU	GSM8K
Individual							
LoRA (no-share)	8	-	-	4.19M	0.06%	35.12	10.84
LoRA (intra-share)	-	8	-	2.10M	0.03%	34.23	10.54
LoRA (inter-share)	-	-	8	0.07M	0.01%	32.20	10.08
Joint							
LoRA (share-intra)	4	4	-	3.14M	0.05%	34.63	**10.92**
LoRA (share-inter)	4	-	4	2.13M	0.03%	35.00	10.54
LoRA (share-intra-inter)	4	2	2	2.64M	0.04%	**35.89**	10.08

Therefore, we combine local parameters and shared parameters to fine-tune the model. Inspired by Wang et al. (2023) and Tian et al. (2024), we employ multiple LoRA modules in parallel to combine local parameters and shared parameters. This method can greatly expand the parameter search space, and enhance the model's adaptability and flexibility. Therefore, the entire set of LoRA parameters

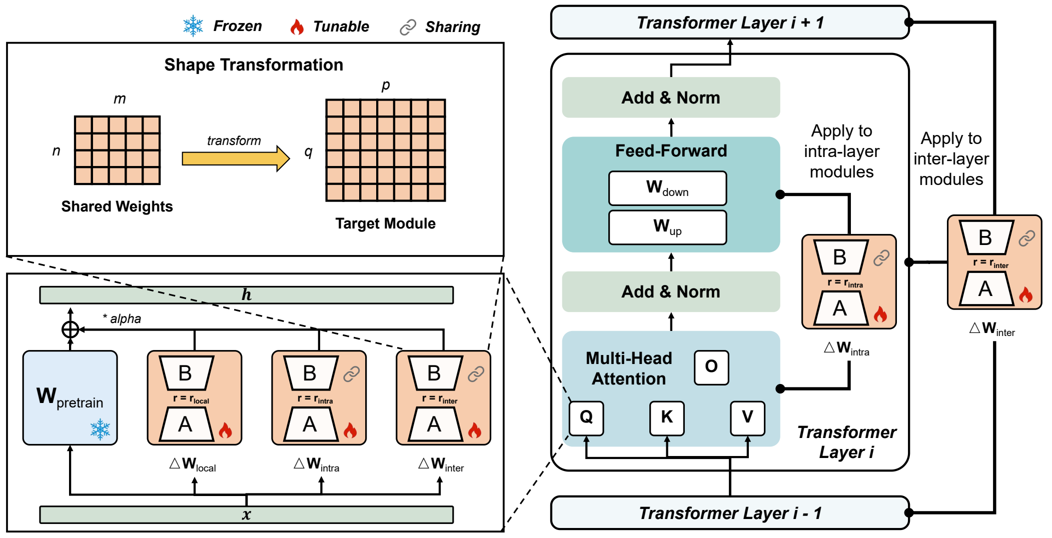

Figure 2: An overview of Bi-Share LoRA. The LoRA weights are decomposed into three sub-LoRA weights: $local$ learns module-specific information, $intra$ capture the shared features within the same transformer block, and $inter$ learns the global information to interact with each module. The shape transformation enables the shared weights adaptive to different shapes of modules. By combining $local$, $intra$, and $inter$, the model can learn both local and global information during fine-tuning, so that improves the performance and generalizability.

is decomposed into three smaller LoRA parameter blocks: the first part acts on individual modules, the second part is shared among modules within the same layer, and the third part is shared by all modules. We assign different ranks for each smaller LoRA, the configuration and results are shown in Table 1 (Joint). The experimental results indicate that sharing parameters not only reduces the number of parameters but also improves model performance.

3 METHOD

3.1 PARAMETER SHARING

Parameter sharing not only facilitates learning global information from the dataset but also significantly reduces the number of parameters in the model (Han et al., 2024). Leveraging this, we apply parameter sharing to enhance parameter efficiency in LoRA fine-tuning. Our approach first decomposes the LoRA module horizontally, expanding the optimization search space and allowing for more efficient learning. We divide the LoRA matrix into three sub-LoRA modules: **local**, **intra-layer**, and **inter-layer** which collectively contribute to the parameter updates of the target module.

Intra-Layer Module refers to the sharing of parameters within the same transformer layer, such that all modules within a single layer (e.g., attention and MLP) share the same LoRA update matrix. This enables the model to capture consistent patterns and correlations within the layer, thereby improving the coherence of the information processed within each layer.

Inter-Layer Module denotes global parameter sharing, where the same LoRA update matrix is shared across different layers of the transformer. This facilitates better information flow and feature interaction between layers, which in turn enhance the overall expressiveness and depth of the model's representations

Local Module refers to the traditional LoRA configuration, where the LoRA update matrix is applied only to the current module, allowing the model to learn highly specific local features. During training and inference, the parameters for each module are updated according to the following equation:

$$f(\boldsymbol{x}) = (\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x} = \boldsymbol{W}\boldsymbol{x} + (\boldsymbol{BA})_{local}\boldsymbol{x} + (\boldsymbol{BA})_{intra}\boldsymbol{x} + (\boldsymbol{BA})_{inter}\boldsymbol{x} \ , \qquad (2)$$

where the distinct rank values of aforementioned three sub-LoRA matrices are denoted as r_{local}, r_{intra}, and r_{inter}, respectively. Specifically, $\boldsymbol{B}_{local} \in \mathbb{R}^{n\times r_{local}}$, $\boldsymbol{A}_{local} \in \mathbb{R}^{r_{local}\times m}$, $\boldsymbol{B}_{intra} \in \mathbb{R}^{n\times r_{intra}}$, $\boldsymbol{A}_{intra} \in \mathbb{R}^{r_{intra}\times m}$, $\boldsymbol{B}_{inter} \in \mathbb{R}^{n\times r_{inter}}$, and $\boldsymbol{A}_{inter} \in \mathbb{R}^{r_{inter}\times m}$.

By introducing this novel decomposition, our Bi-Share LoRA fine-tuning approach is capable of learning both localized features and global interactions, providing a balance between task-specific adaptability and overall model robustness. Figure 2 presents an overview of our Bi-Share LoRA. This design is particularly beneficial in multi-task scenarios, enhancing the model's performance and generalization while significantly reducing the computational resources required during training and inference. By decomposing LoRA into multiple smaller LoRA modules, we can assign a higher rank to the shared parameters, allowing the final parameters to achieve a greater overall rank, as discussed further in Section 4.4.

3.2 Shape Transformation

We initially followed the LoRA setup (Hu et al., 2022) by applying LoRA parameters to the q and v modules in the attention layer, matching the shared parameter size to the qv module, which yielded preliminary results (Table 1). Previous studies suggest that applying LoRA to more modules improves performance (Dettmers et al., 2023), so we extended it to the FFN layers. However, this caused a parameter shape mismatch. In the Transformer block (Vaswani et al., 2017), the $qkvo$ modules in Llama's attention layer have a consistent shape of $(4096, 4096)$. Meanwhile, the FFN's up- and down-projection modules have dimensions of $(4096, 11008)$ and $(11008, 4096)$, respectively, making it difficult to apply a single shared $\boldsymbol{AB}$ parameters across these varying shapes. Similarly, in Llama3, the Grouped-Query Attention (GQA) changes the shape of k module, further complicating the parameter sharing. To address this issue, we develop three transformation methods that adjust shared parameters to size of target modules.

3.2.1 Slice Sharing

The straightforward method is to slice a larger trainable parameter matrix and train only the sliced portions (see Figure 3 (a)), which we refer to as the Slice Sharing (SS) method. Specifically, we determine the maximum input dimension d_{im} and output dimensions d_{om}, among all fine-tuning parameter modules. The shared matrices are then defined with dimensions $\boldsymbol{A}_s \in \mathbb{R}^{r\times d_{im}}$ and $\boldsymbol{B}_s \in \mathbb{R}^{d_{om}\times r}$. During forward computation, the shared matrix is automatically sliced to match the parameter dimensions of the target module. The calculation is expressed as:

$$\Delta \boldsymbol{W} = \boldsymbol{B}_s[:, : d_o]\boldsymbol{A}_s[: d_i, :] \ , \tag{3}$$

where d_i and d_o represent the input and output dimensions of target moddule, $\boldsymbol{B}_s[:, : d_o]$ and $\boldsymbol{A}_s[: d_i, :]$ represent the sliced parts of the shared weights of $\boldsymbol{B}_s$ and $\boldsymbol{A}_s$. Algorithm 1 provides the pseudocode for Slice Sharing.

3.2.2 Gate Transformation

The simple slicing method enables parameter sharing, but only a subset of the shared parameters is used by all modules, while the remaining parameters are only utilized by larger modules. This limits the efficiency of parameter sharing. To address this, we propose matrix multiplication for dynamic size transformation of shared parameters. By multiplying matrices $\boldsymbol{M}_a \in \mathbb{R}^{m\times n}$ and $\boldsymbol{M}_b \in \mathbb{R}^{n\times p}$, the resulting matrix $\boldsymbol{M}_c \in \mathbb{R}^{m\times p}$ transforms the shape. Based on this, we introduce the Gate Transformation (GT), which applies an input gate $\boldsymbol{G}_i \in \mathbb{R}^{m\times d_i}$ and an output gate $\boldsymbol{G}_o \in \mathbb{R}^{d_o\times n}$. For an input $\boldsymbol{x} \in \mathbb{R}^{b\times d_i}$, $\boldsymbol{G}_i$ transforms it to (b, m), and the shared matrix $\boldsymbol{W}_s \in \mathbb{R}^{n\times m}$ processes it to produce an intermediate result (b, n). Finally, $\boldsymbol{G}_o$ outputs the final shape (b, d_o).

However, defining these transformation matrices introduces many learnable parameters, leading to high memory consumption for large inputs and outputs. To mitigate this, we apply one-rank decomposition to the input and output gates, reducing them to the product of two small rank-one matrices (see Figure 3 (b)). The final computation is as follows:

$$\Delta \boldsymbol{W} = \boldsymbol{G}_o\boldsymbol{W}_s\boldsymbol{G}_i = (\boldsymbol{G}_{ou}\boldsymbol{G}_{od})(\boldsymbol{B}_s\boldsymbol{A}_s)(\boldsymbol{G}_{iu}\boldsymbol{G}_{id}) \ , \tag{4}$$

where the $\boldsymbol{G}_{id} \in \mathbb{R}^{1\times d_i}$ projects down the input dimension into low a dimensional space, and then $\boldsymbol{G}_{iu} \in \mathbb{R}^{m\times 1}$ scales it up into the dimension that is comparable to the input dimension of the share

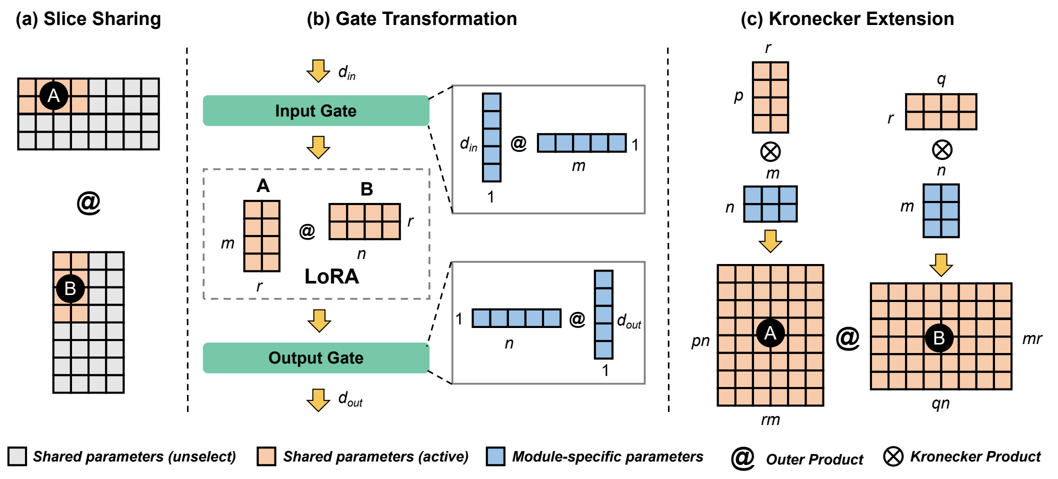

Figure 3: Three methods for shape transformation: (a) Slice Sharing, slices parameters from a large shared trainable parameters; (b) Gate Transformation, an input gate and an output gate transforms input dimension and output dimension to match the shape of shared parameters, and they are implemented by one-rank decomposition; (c) Kronecker Extension, a module-specific kernel are used to extend the shared weights into target shape through the Kronecker Product.

weights $\boldsymbol{W}_s$. Similarly, $\boldsymbol{G}_{od} \in \mathbb{R}^{1\times n}$ and $\boldsymbol{G}_{ou} \in \mathbb{R}^{d_o\times 1}$ are applied to the transform the output. By setting the input and output gates, the size of our shared parameters can be flexibly changed. Algorithm 2 provides the pseudocode for Gate Transformation.

3.2.3 Kronecker Extension

While utilizing Gate Transformation allows us to define shared parameters of arbitrary shapes, low-rank decomposition may lead to information loss in both input and output transformations. Another approach is to concatenate multiple small shared parameters from identical copies to form a larger shared parameter (Wang et al., 2024; Edalati et al., 2022), but this limits the overall expressiveness. To address this, inspired by Karimi Mahabadi et al. (2021), we apply Kronecker matrix multiplication to expand the dimensions of the shared matrix by integer multiples, a method we term Kronecker Extension (KE). The Kronecker product between matrices $\boldsymbol{X} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{p\times q}$, denoted as $\boldsymbol{X} \otimes \boldsymbol{Y} \in \mathbb{R}^{mp\times nq}$, is mathematically defined as:

$$\boldsymbol{X} \otimes \boldsymbol{Y} = \begin{pmatrix} x_{11}\boldsymbol{Y} & \cdots & x_{1f}\boldsymbol{Y} \\ \vdots & \ddots & \vdots \\ x_{m1}\boldsymbol{Y} & \cdots & x_{mf}\boldsymbol{Y} \end{pmatrix} , \tag{5}$$

where x_{ij} shows the element in the i^{th} row and j^{th} column of $\boldsymbol{X}$.

We assign a module-specific kernel $\boldsymbol{K} \in \mathbb{R}^{d\times r}$ to each module in the model (see Figure 3(c)). By applying the Kronecker product, we expand the shared parameter $\boldsymbol{A}_s \in \mathbb{R}^{r\times \frac{m}{k}}$ and $\boldsymbol{B}_s \in \mathbb{R}^{\frac{n}{k}\times r}$ to match the size $\boldsymbol{A}_t \in \mathbb{R}^{r\times m}$ and $\boldsymbol{B}_t \in \mathbb{R}^{n\times r}$ of the target module. Here, r is the rank value set for the shared parameter. Finally, according to our Kronecker Extension, $\Delta\boldsymbol{W}$ is calculated as:

$$\Delta\boldsymbol{W} = (\boldsymbol{K}_B \otimes \boldsymbol{B}_s)(\boldsymbol{K}_A \otimes \boldsymbol{A}_s) , \tag{6}$$

where $\boldsymbol{K}_B \in \mathbb{R}^{k\times 1}$ and $\boldsymbol{K}_A \in \mathbb{R}^{1\times k}$ represent module-specific kernel for the $\boldsymbol{B}$ and $\boldsymbol{A}$ matrices in the LoRA module. Algorithm 3 provides the pseudocode for Kronecker Extension.

4 Experiments

4.1 Settings

LLMs. To demonstrate how Bi-Share LoRA performed on different models, we conduct experiments on Llama families (Touvron et al., 2023): Llama 1 and Llama 3. In particular, we fine-tune

Table 2: Results of Zero-shot performance on Llama 1-7B, Llama 3-8B, and Llama 1-13B in Bi-Share LoRA and baselines on Commonsense Reasoning benchmark. We report the number of trainable parameters (# params) and the corresponding ratio for each method.

	Methods	# params	ratio	OBQA	ARC-c	HellaSwag	ARC-e	PIQA	WinoG.	BoolQ	SIQA	Avg.
Llama 1-7B	LoRA$_{r=8}$	14.02M	0.21%	44.80	47.10	$\underline{77.35}$	76.47	$\underline{80.25}$	69.77	77.98	48.21	65.24
	VeRA$_{r=64}$	0.89M	0.01%	44.60	44.80	76.47	75.88	79.27	$\underline{70.24}$	75.14	46.37	64.09
	VB-LoRA$_{r=4}$	2.49M	0.04%	$\underline{46.00}$	47.35	77.28	$\underline{77.44}$	79.98	**70.48**	76.79	48.77	65.51
	Bi-Share-LoRA (SS)	7.03M	0.10%	**46.20**	46.93	77.23	76.77	**80.52**	69.85	$\underline{78.13}$	**49.39**	$\underline{65.63}$
	Bi-Share-LoRA (GT)	8.22M	0.12%	45.20	$\underline{47.27}$	**77.47**	77.06	80.14	70.17	**78.93**	$\underline{48.93}$	**65.64**
	Bi-Share-LoRA (KE)	3.66M	0.05%	45.40	**47.87**	77.32	**77.57**	80.20	$\underline{70.24}$	77.52	48.62	65.59
Llama 3-8B	LoRA$_{r=8}$	14.16M	0.18%	$\underline{46.20}$	**57.34**	**80.04**	$\underline{82.95}$	81.88	73.72	82.32	48.67	72.42
	VeRA$_{r=64}$	0.80M	0.01%	45.00	54.01	79.27	80.51	81.23	73.32	81.07	47.34	70.96
	VB-LoRA$_{r=4}$	2.51M	0.03%	**46.40**	56.06	79.85	81.27	81.39	$\underline{74.51}$	81.62	46.93	71.66
	Bi-Share-LoRA (SS)	7.67M	0.10%	**46.40**	$\underline{57.17}$	$\underline{79.96}$	$\underline{82.95}$	$\underline{81.94}$	**74.74**	**83.09**	**49.03**	**72.70**
	Bi-Share-LoRA (GT)	8.03M	0.10%	46.20	56.83	79.89	82.87	$\underline{81.94}$	74.27	$\underline{82.97}$	48.36	$\underline{72.45}$
	Bi-Share-LoRA (KE)	3.83M	0.05%	**46.40**	56.57	**80.04**	**83.08**	**82.15**	73.64	82.60	$\underline{48.98}$	72.44
Llama 1-13B	LoRA$_{r=8}$	21.95M	0.17%	45.40	$\underline{51.71}$	80.21	**79.21**	**80.90**	$\underline{72.69}$	$\underline{81.13}$	48.87	67.52
	VeRA$_{r=128}$	1.40M	0.01%	44.80	47.87	79.30	77.61	80.25	**72.85**	78.07	46.88	65.95
	VB-LoRA$_{r=8}$	3.88M	0.04%	**47.20**	51.11	**80.91**	78.66	80.58	72.38	80.18	**49.49**	$\underline{67.56}$
	Bi-Share-LoRA (SS)	10.13M	0.08%	$\underline{45.80}$	51.28	80.11	**79.21**	$\underline{80.74}$	$\underline{72.69}$	**81.59**	$\underline{49.13}$	**67.57**
	Bi-Share-LoRA (GT)	12.14M	0.9%	46.00	51.02	80.11	$\underline{79.04}$	**80.90**	72.53	80.95	48.72	67.41
	Bi-Share-LoRA (KE)	5.94M	0.05%	45.00	**51.79**	$\underline{80.32}$	**79.21**	$\underline{80.74}$	**72.85**	80.83	48.93	67.46

the 7B and 13B models of Llama 1 and 8B model of Llama 3, with the specific versions detailed in Appendix A.13.

Benchmark. We conduct experiments for these LLMs on two different benchmarks. The first benchmark is **Commonsense Reasoning**, which includes BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2021), ARC-easy (Clark et al., 2018), ARC-challenge (Clark et al., 2018), OpenbookQA (Mihaylov et al., 2018), and SIQA (Sap et al., 2019). The second benchmark is **Massively Multitask Language Understanding (MMLU)** (Hendrycks et al., 2021). Dataset details are presented in Appendix A.14. We employed lm-eval-harness (Gao et al., 2023) to create open prompts for the benchmarks and produce the results.

Baselines. We compare against several recently proposed LoRA-based PEFT methods: **(1) LoRA** (Hu et al., 2022), we set the rank to 8 for the standard LoRA to fine-tune the model. **(2) VeRA** (Kopiczko et al., 2024), we adopt the default setting where the rank is set as 64. **(3) VB-LoRA** Li et al. (2024), we follow the setting of VB-LoRA, where the vector length is set to 256 and there are 90 vectors to be trained. Moreover, we set the k of the top-k to 2.

Fine-tuning Dataset. We utilized publicly available samples from the Alpaca dataset (Taori et al., 2023) [1] to further fine-tune the LLM, which contains 52k instruction-following demonstrations generated by OpenAI's text-davinci-003 engine.

Hyper-parameters and Training Details. We apply the LoRA weights to the $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, $\boldsymbol{W}_v$, $\boldsymbol{W}_{up}$, and $\boldsymbol{W}_{down}$ modules of each Transformer block. For each shape transformation method, we set different rank configurations $r = \{r_{local}, r_{intra}, r_{inter}\}$, We set $r = \{2, 4, 32\}$ for Slice Share (SS). We set the shape of shared weights for Gate Transformation (GT) to $(1024, r_{share})$ and $r = \{2, 8, 16\}$. For the Kronecker Extension (KE), we set $r = \{2, 4, 16\}$ to adapt the shared weights' shape of $(256, r_{share})$. We use the same training configurations to fine-tune the LLM with Bi-Share LoRA and baseline methods. Specifically, we use AdamW (Loshchilov, 2017) as the optimizer with 100 warm-up steps and a learning rate of 1×10^{-4} and set the batch size to 64. For all the experiments, we train for one epoch.

4.2 Results on Commonsense Reasoning

We evaluate the zero-shot performance of Bi-Share LoRA on Commonsense Reasoning tasks using Llama 1-7B, Llama 3-8B, and Llama 1-13B models. In Table 2, the results show that Bi-Share LoRA consistently outperforms the baselines in terms of average performance across these datasets.

[1] `https://huggingface.co/datasets/yahma/alpaca-cleaned`

Table 3: Results of zero-shot and five-shot performance on Llama 1-7B, Llama 3-8B, and Llama 1-13B in Bi-Share LoRA and baselines on MMLU benchmark. We report the number of trainable parameters (# params) and the corresponding ratio for each method.

	Method	# params	ratio	MMLU (0-shot)					MMLU (5-shot)				
				Hums.	STEM	Social	Other	Avg.	Hums.	STEM	Social	Other	Avg.
Llama 1-7B	LoRA$_{r=8}$	14.02M	0.21%	34.67	31.24	37.21	39.36	35.62	34.86	32.57	40.36	40.81	37.15
	VeRA$_{r=64}$	0.89M	0.01%	32.22	28.32	32.40	36.72	32.42	33.58	31.27	38.58	38.85	35.57
	VB-LoRA$_{r=4}$	2.49M	0.04%	30.10	28.51	29.25	33.31	30.29	34.11	30.54	40.14	40.52	36.33
	Bi-Share-LoRA (SS)	7.03M	0.10%	**36.34**	**32.98**	40.59	**42.55**	**38.12**	**36.62**	**33.46**	**42.48**	**43.35**	**38.98**
	Bi-Share-LoRA (GT)	8.22M	0.12%	36.30	32.92	**40.75**	42.13	38.03	35.15	32.19	41.44	41.94	37.68
	Bi-Share-LoRA (KE)	3.66M	0.05%	35.56	32.00	38.12	40.01	36.42	35.32	32.41	41.40	41.29	37.61
Llama 3-8B	LoRA$_{r=8}$	14.16M	0.18%	56.77	53.89	72.77	70.36	63.44	59.81	55.92	76.21	72.10	66.01
	VeRA$_{r=64}$	0.80M	0.01%	54.88	54.17	**73.35**	70.58	63.25	**59.85**	55.69	76.15	**72.80**	66.12
	VB-LoRA$_{r=4}$	2.51M	0.03%	55.83	52.74	71.86	71.00	62.86	59.11	55.22	74.68	72.26	65.32
	Bi-Share-LoRA (SS)	7.67M	0.10%	**58.53**	**54.61**	**73.35**	71.29	**64.45**	59.57	56.26	75.95	72.35	66.04
	Bi-Share-LoRA (GT)	8.03M	0.10%	58.30	54.39	73.29	**71.48**	64.37	59.68	56.23	**76.31**	72.48	**66.18**
	Bi-Share-LoRA (KE)	3.83M	0.05%	57.98	54.49	**73.35**	70.84	64.16	59.38	**55.98**	76.08	72.16	65.90
Llama 1-13B	LoRA$_{r=8}$	21.95M	0.17%	43.66	38.31	**54.44**	53.33	47.43	45.06	37.46	**55.64**	55.42	48.39
	VeRA$_{r=128}$	1.40M	0.01%	41.66	36.50	48.75	48.73	43.91	44.23	37.08	53.92	53.59	47.20
	VB-LoRA$_{r=8}$	3.88M	0.04%	41.02	35.81	49.66	49.76	44.06	44.27	37.14	54.27	53.91	47.40
	Bi-Share-LoRA (SS)	10.13M	0.08%	**44.76**	37.46	53.04	53.72	47.24	44.78	37.81	54.50	**55.65**	48.18
	Bi-Share-LoRA (GT)	12.19M	0.09%	43.61	37.77	53.33	53.14	46.96	44.46	37.52	53.88	55.07	47.73
	Bi-Share-LoRA (KE)	5.94M	0.05%	44.48	**38.98**	54.34	**54.49**	**48.07**	**45.29**	**37.84**	55.25	55.36	**48.43**

Specifically, the Kronecker Extension (KE) method introduces fewer trainable parameters while achieving comparable performance to the SS and GT methods, indicating its superior parameter efficiency. More detailed results can be found in Appendix A.5.1.

4.3 Results on MMLU Benchmark

We evaluate the zero-shot and five-shot performance of Bi-Share LoRA on the MMLU benchmark using Llama 1-7B, Llama 3-8B, and Llama 1-13B models. The results, as shown in Table 3, demonstrate that Bi-Share LoRA consistently outperforms the baseline models in terms of average performance across both zero-shot and five-shot settings. This highlights the effectiveness of our approach in diverse scenarios (see more in Appendix A.5.2).

4.4 Analysis

Rank Analysis. According to matrix rank theory, the rank of the sum of two matrices is given by $\mathcal{R}(\boldsymbol{A}+\boldsymbol{B}) \leq \mathcal{R}(\boldsymbol{A}) + \mathcal{R}(\boldsymbol{B})$. This implies that decomposing a large LoRA matrix into multiple sub-LoRAs does not increase the overall rank of the matrix. Consequently, the rank of the combined LoRA matrices remains bounded by the sum of their individual ranks. We validate the actual rank of each module and calculate the average ranks across layers for each Shape Transformation method using rank configurations of $r = \{2, 4, 16\}$. The results, shown in Figure 4(a), indicate that the Slice Sharing (SS) method achieves a combined rank equal to the sum of local and shared ranks (22), while the Kronecker Extension (KE) reaches approximately 21.53. In contrast, the Gate Transformation (GT) method yields rank value equivalent to the local rank plus 2, likely due to one-rank gates causing some information loss.

Contribution Analysis. We conduct further experiments to explore the contribution of each sub-LoRA matrix. By setting the rank of one sub-LoRA matrix to 8 and the others to 0, we examine the individual impact of each component. The Kronecker Extension method is used to reshape the shared parameters. In Figure 4(b), the results reveal the performance preferences of each component across datasets. Specifically, the local component of LoRA performs best on HellaSwag, ARC-e, BoolQ, and SIQA, while the intra-layer shared component excels on PIQA and ARC-c. Overall, the combination of local, intra-layer, and inter-layer shared parameters yields the best performance across all datasets.

Extension Analysis. We further analyze the scalability of the Kronecker Extension method. The shared matrix $\boldsymbol{W} \in \mathbb{R}^{n \times r}$ can be obtained by applying the Kronecker product to $\boldsymbol{M} \in \mathbb{R}^{\frac{n}{k} \times r}$ and $\boldsymbol{K} \in \mathbb{R}^{k \times 1}$. By adjusting their shapes to $\boldsymbol{M} \in \mathbb{R}^{\frac{n}{k} \times 1}$ and $\boldsymbol{K} \in \mathbb{R}^{k \times r}$, the same matrix shape for $\boldsymbol{W}$ can be achieved. Furthermore, since one dimension is a constant 1, the resulting matrix rank equals r. To explore its effect, we modify this constant to 2 to test whether it enhances the rank of $\boldsymbol{W}$. We denote configurations as a_b, where r_1 means the shared matrix has a dimension r and a module-specific dimension of 1, while 2_r has a shared dimension of 2 and a module-specific dimension of r. The results in Figure 4(c) show that modifying the constant improves both expressiveness and information content. Comparing r_1 and 1_r, we observe that each configuration excels in different metrics, with r_1 performing better overall while introducing fewer trainable parameters.

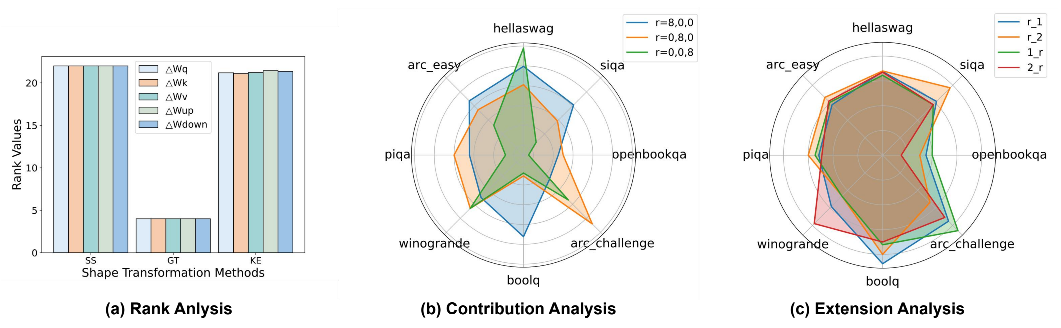

Figure 4: Analysis. (a) **Rank Analysis**: the final rank benefit from different Shape Transformation methods. (b) **Contribution Analysis**: the different contribution across the sharing weights and local weights. (c) **Extension Analysis**: the rank extension analysis for different shape of share weights' size and kernel size.

4.5 Ablation Study

We conduct ablation studies to examine the impact of individual components of our method. All subsequent experiments focus on the MMLU (zero-shot) (Hendrycks et al., 2021) and GSM8K (5-shot) (Cobbe et al., 2021) benchmarks, utilizing the Llama 1-7B model. We also conduct experiments to examine the impact of different rank allocation in Appendix A.10.

Slicing Method. We evaluate three slicing methods for the shared matrix: top-left slice, bottom-right slice, and center slice. We adopt the rank configuration of $r = \{2, 4, 8\}$. The results are shown in Table 4. It indicates that the center slice outperforms the other two slice methods both on MMLU and GSM8K benchmark.

Gate Initialization. We initialize the input and output gates of the Gate Transformation using three schemes: Kaiming normal, Kaiming uniform, and constant one initialization. The rank configurations for different sub-LoRA weights are set for $r = \{2, 8, 16\}$. As shown in Table 5, Kaiming uniform outperforms Kaiming normal initialization. Additionally, the constant ones initialization leads to gradient explosion or vanishing issues, making it unsuitable for gate initialization.

Initialization of Kronecker Kernel. We apply Kaiming normal, Kaiming uniform, and constant ones initialization to the Kronecker kernel to examine the impact of different initialization schemes. We set the ranks of $\{2, 4, 16\}$ for local, intra, and inter sub-LoRA matrices, respectively. From Table 6, the constant ones initialization performs better on the MMLU benchmark, while the Kaiming normal initialization outperforms the other two methods on the GSM8K benchmark. Overall, the Kaiming normal initialization performs best.

Table 4: Split Position

Method	MMLU	GSM8K
Top-Left	36.27	10.69
Right-Down	35.84	10.92
Center	**36.58**	**11.14**

Table 5: Gate Initialization

Matrix Init.	MMLU	GSM8K
Kaiming Unif.	**36.36**	**11.90**
Kaiming Norm.	36.09	10.31
Ones	NAN	NAN

Table 6: Kernel Initialization

Matrix Init.	MMLU	GSM8K
Kaiming Unif.	35.34	11.14
Kaiming Norm.	35.15	**12.05**
Ones	**35.84**	11.14

Shared Size of Gate and Kronecker kernel. The Gate Transformation (GT) and Kronecker Extension (KE) methods can flexibly adapt the shared weights to arbitrary shapes for different modules. We examine various sizes of the shared matrix, setting the rank $r = \{2, 8, 16\}$. For the GT, we test shared matrix sizes of [512, 1024, 2048], while for the KE, we test sizes of [64, 128, 256]. The results are presented in Table 7 and Table 8. We found that the parameters introduced by GT increase with the growth of the shared parameter size, which also leads to an improvement in performance on the MMLU benchmark. However, we observe a decrease in performance on the GSM8K dataset as the shared parameter size increases. In contrast, the parameters introduced by the KE method decrease with increasing shared parameter size, while the average performance improves. Overall, we select the shared parameter size of 1024 for the GT method and 256 for the KE method.

Table 7: Size of Shared weights of GT

Shared Size	# params	raito	MMLU	GSM8K
512	7.62M	0.11%	**36.03**	10.54
1,024	8.22M	0.12%	35.61	10.77
2,048	9.44M	0.13%	34.70	**11.90**

Table 8: Size of Shared weights of KE

Shared Size	# params	raito	MMLU	GSM8K
64	4.07M	0.06%	36.47	10.84
128	3.82M	0.05%	35.34	**11.98**
256	3.72M	0.05%	**37.01**	11.22

5 Related Work

5.1 Multi-LoRA Architecture

LoRA has demonstrated exceptional resource efficiency and performance in adapting LLMs for specific tasks, driving the demand for a single model capable of handling multiple tasks (Agiza et al., 2024). Several approaches have been proposed to improve their multi-task adaptability. In particular, LoraHub (Huang et al., 2023) assembles LoRA modules trained on different tasks to eliminate the need for human expertise and assumptions, enabling effective cross-task generalization. Similarly, MultiLoRA (Wang et al., 2023) improves adaptability by horizontally expanding LoRA modules and reducing the dominance of top singular vectors observed in LoRA. Building on these advancements, HydraLoRA (Tian et al., 2024) introduces an asymmetric architecture that shares a common matrix across tasks while using task-specific matrices for different sub-domains, further enhancing both fine-tuning and inference efficiency.

5.2 Parameter Sharing of Lora

Recent advances in LoRA-based fine-tuning methods have explored various strategies for sharing LoRA weights to enhance parameter efficiency across multiple tasks. VeRA (Kopiczko et al., 2024) proposes sharing random matrices across all layers, reducing the number of parameters, but it results in some performance trade-offs and increased inference latency due to its high-rank requirements. In addition, Tied-LoRA (Renduchintala et al., 2024) takes a different approach by sharing LoRA matrices specifically across query, key, and value projection layers, using additional scaling vectors to differentiate the modules. However, its requirement for identical matrix shapes limits flexibility. In contrast, PRoLoRA (Wang et al., 2024) employs an intra-layer sharing mechanism with learnable parameters, but it only reduces parameters without capturing global features. Additionally, VB-LoRA (Li et al., 2024) introduces a "divide-and-share" approach that partitions shared vectors into a vector bank, addressing the limitations of traditional low-rank decomposition across matrix dimensions, modules, and layers. However, this approach selects vectors uniformly without accounting for the internal structure of the model, leading to suboptimal utilization of the model's internal information

6 Conclusion

This paper presents Bi-Share LoRA, a method that enhances the parameter efficiency of large language models (LLMs) by combining shared intra-layer, inter-layer parameters, and local parameters. This approach reduces the number of trainable parameters while boosting model efficiency. Experiments on various Llama models demonstrate that Bi-Share LoRA significantly cuts down parameter usage by 56.40% and improves average performance by 0.33% on commonsense reasoning tasks and 2.08% on MMLU benchmark. Overall, Bi-Share LoRA effectively reduces redundancy and enhances model adaptability across diverse tasks. In the future, we will explore more intelligent parameter sharing modes that can selectively select different modules or different layers for parameter sharing, ultimately further improving the performance of the model.

7 Ethics Statement

This paper is built upon pre-trained large language models (e.g., Llama1 and Llama3) and existing datasets for instruct fine-tuning (e.g., Alpaca). We do not introduce any new data and thus do not involve human annotation. This paper has no additional ethical concerns beyond a large corpus of research in LLMs.

8 Reproducibility Statement

We have thoroughly elucidated our design and training details throughout the paper to provide a comprehensive understanding of our methodology. Specifically, the overall sharing architecture is described in Section 3.1, where we outline how parameters are effectively shared across different layers and modules. The Shape Transformation methods are detailed in Section 3.2, illustrating the various techniques employed to adapt shared weights to different configurations. Additionally, we provide the corresponding pseudocode for these methods in Appendix A.3, allowing readers to easily follow the implementation details. Furthermore, the hardware and environment used for our experiments are specified in Appendix A.13, including the specifications of the computing resources and software versions that facilitated our experiments. We also list the specific versions of the large language models (LLMs) utilized in Appendix A.12, ensuring clarity regarding the models' configurations. The training datasets and evaluation benchmarks, crucial for assessing model performance, are detailed in Section 4.1 and Appendix A.14. This includes a thorough description of the datasets employed for both training and evaluation phases. Finally, we outline the hyperparameters and configurations in Section 4.1, providing insight into the settings that governed the training process. By compiling these details, we aim to enhance the reproducibility of our work and assist other researchers in understanding and applying our methods effectively.

References

Ahmed Agiza, Marina Neseem, and Sherief Reda. Mtlora: Low-rank adaptation approach for efficient multi-task learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16196–16205, 2024.

Anthropic. The claude 3 model family: Opus, sonnet, haiku. 2024. URL `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2924–2936, 2019.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Proceedings of Advances in Neural Information Processing Systems*, volume 36, pp. 10088–10115, 2023.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235, 2023.

Ali Edalati, Marzieh Tahaei, Ivan Kobyzev, Vahid Partovi Nia, James J. Clark, and Mehdi Rezagholizadeh. Krona: Parameter efficient tuning with kronecker adapter, 2022.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 2023.

Chaoyu Guan, Xiting Wang, Quanshi Zhang, Runjin Chen, Di He, and Xing Xie. Towards a deep and unified understanding of deep neural models in NLP. In *Proceedings of the International Conference on Machine Learning*, volume 97, pp. 2454–2463, 2019.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the International Conference on Machine Learning*, volume 97, pp. 2790–2799, 2019.

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 *Proceedings of International Conference on Learning Representations*, 2022.

Chengsong Huang, Qian Liu, Bill Yuchen Lin, Chao Du, Tianyu Pang, and Min Lin. Lorahub: Efficient cross-task generalization via dynamic loRA composition. In *Proceedings of R0-FoMo:Robustness of Few-shot and Zero-shot Learning in Large Foundation Models*, 2023.

Ting Jiang, Shaohan Huang, Shengyue Luo, Zihan Zhang, Haizhen Huang, Furu Wei, Weiwei Deng, Feng Sun, Qi Zhang, Deqing Wang, and Fuzhen Zhuang. Mora: High-rank updating for parameter-efficient fine-tuning, 2024.

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. In *Advances in Neural Information Processing Systems*, volume 34, pp. 1022–1035, 2021.

Soroush Abbasi Koohpayegani, Navaneet K L, Parsa Nooralinejad, Soheil Kolouri, and Hamed Pirsiavash. NOLA: Compressing loRA using linear combination of random basis. In *The Twelfth International Conference on Learning Representations*, 2024.

Dawid Jan Kopiczko, Tijmen Blankevoort, and Yuki M Asano. VeRA: Vector-based random matrix adaptation. In *Proceedings of International Conference on Learning Representations*, 2024.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2021.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*, pp. 4582–4597, 2021.

Yang Li, Shaobo Han, and Shihao Ji. Vb-lora: Extreme parameter efficient fine-tuning with vector banks, 2024.

Sihao Lin, Pumeng Lyu, Dongrui Liu, Tao Tang, Xiaodan Liang, Andy Song, and Xiaojun Chang. Mlp can be a good transformer learner. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 19489–19498, 2024.

I Loshchilov. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Yuren Mao, Yuhang Ge, Yijiang Fan, Wenyi Xu, Yu Mi, Zhonghao Hu, and Yunjun Gao. A survey on lora of large language models. *arXiv preprint arXiv:2407.11046*, 2024.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2381–2391, 2018.

Openai. Gpt-4 technical report. 2023. URL `https://cdn.openai.com/papers/gpt-4.pdf`.

Mohaimenul Azam Khan Raiaan, Md Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access*, 2024.

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 Annual Meeting of the Association for Computational Linguistics*, pp. 3052–3064, 2024.

Adithya Renduchintala, Tugrul Konuk, and Oleksii Kuchaiev. Tied-lora: Enhancing parameter efficiency of lora with weight tying, 2024.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 4463–4473, 2019.

Shuhua Shi, Shaohan Huang, Minghui Song, Zhoujun Li, Zihan Zhang, Haizhen Huang, Furu Wei, Weiwei Deng, Feng Sun, and Qi Zhang. Reslora: Identity residual mapping in low-rank adaption, 2024.

Justin Sirignano and Konstantinos Spiliopoulos. Mean field analysis of neural networks: A law of large numbers. *SIAM Journal on Applied Mathematics*, 80(2):725–752, 2020.

Zehua Sun, Huanqi Yang, Kai Liu, Zhimeng Yin, Zhenjiang Li, and Weitao Xu. Recent advances in lora: A comprehensive survey. *ACM Transactions on Sensor Networks*, 18(4):1–44, 2022a.

Zhenhong Sun, Ce Ge, Junyan Wang, Ming Lin, Hesen Chen, Hao Li, and Xiuyu Sun. Entropy-driven mixed-precision quantization for deep network design. In *Proceedings of Advances in Neural Information Processing Systems*, volume 35, pp. 21508–21520, 2022b.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Stanford alpaca: An instruction-following llama model, 2023. URL `https://github.com/tatsu-lab/stanford_alpaca`.

Chunlin Tian, Zhan Shi, Zhijiang Guo, Li Li, and Chengzhong Xu. Hydralora: An asymmetric lora architecture for efficient fine-tuning, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems*, volume 30, 2017.

Sheng Wang, Boyang Xue, Jiacheng Ye, Jiyue Jiang, Liheng Chen, Lingpeng Kong, and Chuan Wu. Prolora: Partial rotation empowers more parameter-efficient lora, 2024.

Yiming Wang, Yu Lin, Xiaodong Zeng, and Guannan Zhang. Multilora: Democratizing lora for better multi-task learning, 2023.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement, 2024.

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 *Forty-first International Conference on Machine Learning*, 2024.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, 2019.

Biao Zhang, Barry Haddow, and Alexandra Birch. Prompting large language model for machine translation: A case study. In *Proceedings of International Conference on Machine Learning*, pp. 41092–41110, 2023.

A APPENDIX

A.1 PARAMETER-EFFICIENT FINE-TUNING (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) is crucial for large language models (LLMs) as it reduces computational costs while preserving performance (Ding et al., 2023). Specifically, Adapters (Houlsby et al., 2019), inserted between layers, allow task-specific fine-tuning with minimal trainable parameters while keeping most weights fixed. Similarly, Prefix Tuning (Li & Liang, 2021) adds learnable tokens at each transformer layer to guide task-specific behavior without modifying core parameters. In contrast, Prompt Tuning (Lester et al., 2021) optimizes a small set of prompts attached to the input, leaving the model's architecture unchanged. LoRA (Hu et al., 2022) reparameterizes weight matrices into low-rank forms, significantly reducing trainable parameters while maintaining performance. Some studies build upon LoRA to further improve and optimize its performance (Shi et al., 2024; Ren et al., 2024; Jiang et al., 2024; yang Liu et al., 2024; Koohpayegani et al., 2024). Likewise, we propose Bi-Share LoRA, building on LoRA, which shares parameters both within and across layers to further enhance efficiency and adaptability.

A.2 PARAMETER SIMILARITY

A.2.1 ENTROPY QUANTIFICATION

Information entropy is a key concept in information theory, used to quantify the uncertainty or randomness within a dataset or signal. It provides a measure of the average information content per symbol or event in a message or sequence of messages. As defined by Guan et al. (2019), entropy can be employed to assess the information capacity of a network. Consequently, the entropy of a given layer can be calculated based on the probability distribution of its features:

$$H(F) = -\int p(f) log p(f) df, f \in F \quad . \tag{7}$$

Nonetheless, it is difficult to directly measure the probability distribution of a feature map: $p(f), f \in F$. Following (Sirignano & Spiliopoulos, 2020; Sun et al., 2022b), we use the Gaussian distribution as the probability distribution of the intermediate feature in a layer. Therefore, the entropy of a certain layer is approximated as the mathematical expectation of $F \sim \mathcal{N}(\mu, \sigma^2)$:

$$\begin{aligned} H(F) &= -\mathbb{E}\left[\log \mathcal{N}\left(\mu, \sigma^2\right)\right] \\ &= -\mathbb{E}\left[\log\left[\left(2\pi\sigma^2\right)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}(f-\mu)^2\right)\right]\right] \\ &= \log(\sigma) + \frac{1}{2}\log(2\pi) + \frac{1}{2} \quad , \end{aligned} \tag{8}$$

where σ is the standard deviation of the feature set $f \in F$.

A.2.2 ENTROPY SIMILARITY

The information content of LoRA weight matrices can be measured by their entropy. Two matrices with similar entropy values typically contain similar amounts of information, indicating a high degree of redundancy. We assess the correlation between different weight matrices by calculating mutual information, which quantifies the relationship between them and can be computed using the following formula:

$$I(\boldsymbol{X};\boldsymbol{Y}) = H(\boldsymbol{X}) + H(\boldsymbol{Y}) - H(\boldsymbol{X},\boldsymbol{Y}) \quad , \tag{9}$$

Here, $H(\boldsymbol{X},\boldsymbol{Y})$ represents the entropy of the joint distribution of matrix $\boldsymbol{X}$ and matrix $\boldsymbol{Y}$. To compute this, we flatten the matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, concatenate them, and then calculate the entropy of the resulting joint distribution. A large mutual information between two weight matrices indicates significant overlap in their information, implying redundancy. The calculated mutual information $I(\boldsymbol{X};\boldsymbol{Y})$ is an absolute value, which needs to be converted into a relative value. Therefore, we use Relative Mutual Information (RMI) to represent the similarity between matrices, calculated as follows:

$$RMI = \frac{I(\boldsymbol{X};\boldsymbol{Y})}{min(H(\boldsymbol{X}), H(\boldsymbol{Y}))} \tag{10}$$

If $RMI > 0.8$, the mutual information is considered high, indicating significant redundancy between the two matrices. If $0.5 < RMI \leq 0.8$, the mutual information is moderate, suggesting notable shared information but with some degree of independence. If $RMI \leq 0.5$, the mutual information is low, indicating minimal redundancy between the two matrices.

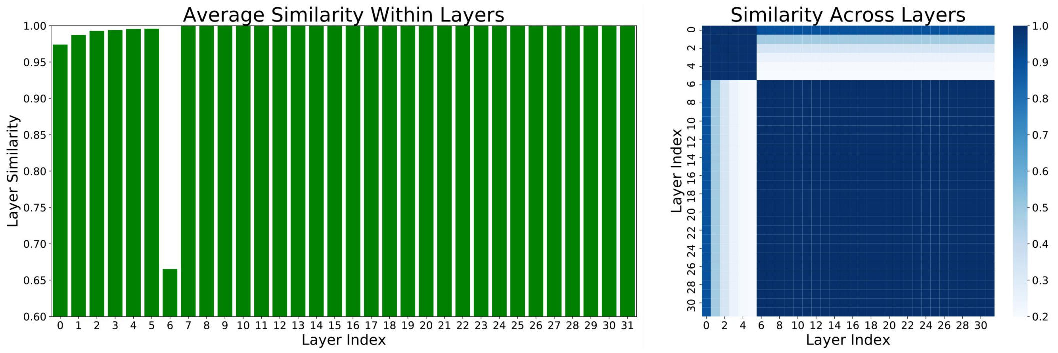

Figure 5: The entropy similarity of LoRA parameters for each module within the same layer (left) and across different layers (right). It shows that different modules within the same layer exhibit high entropy similarity, and this high similarity is also present across different layers. This indicates that LoRA parameters have a significant degree of redundancy.

We fine-tuned the LLaMA-7B model using LoRA with a rank $r = 64$, isolating the LoRA matrices and calculating $\Delta \boldsymbol{W}$. The information entropy of $\Delta \boldsymbol{W}$ for each module was computed based on Equation 8, and the Relative Mutual Information (RMI) between modules was calculated as a measure of similarity using Equations 9 and 10. We obtained the RMI values both within the same layer and across different layers, with the results shown in Figure 1. We observed that different modules within the same layer exhibit high entropy similarity, and this similarity also extends across layers. Additionally, following the method in Lin et al. (2024), we calculated the entropy similarity of the activation values for each module. As illustrated in Figure 5, the $\Delta \boldsymbol{W}$ of different parameter modules also show high similarity in terms of activation values, further indicating the significant redundancy in the parameters used in LoRA fine-tuning.

A.3 Shape Transformation Algorithms

We provide detailed pseudocode for each of the Shape Transformation methods used in Bi-Share LoRA. Specifically, Algorithm 1 outlines the steps for the Slice Sharing method, which slices larger matrices to adapt to different parameter dimensions. Algorithm 2 demonstrates the Gate Transformation method, which utilizes input and output gates to dynamically adjust shared parameters for varying module shapes. Finally, Algorithm 3 explains the Kronecker Extension method, which expands shared matrices using Kronecker products to maintain consistency across modules with diverse dimensions. These algorithms collectively contribute to enhancing parameter efficiency and flexibility in model fine-tuning.

A.4 Parameter Count Analysis

In this section, we compare the number of parameters of Bi-Share LoRA with the standard LoRA. Considering an LLM with L layers, where each layer contains M modules with hidden dimension d, the number of trainable parameters is equal to the model size (i.e., LMd^2) in full fine-tuning. LoRA reduces this number to $2LMdr$, where r is the rank of two low-rank decomposed matrices. In Bi-Share LoRA, the trainable parameters consist of two parts: local parameters $\mathcal{L}$, computed as $2LMdr_{local}$, and shared parameters, which include intra-parameter sharing $\mathcal{S}intra$ and inter-parameter sharing $\mathcal{S}inter$. Therefore, the stored parameters can be represented by a triplet $\Theta = \{\mathcal{L}, \mathcal{S}_{intra}, \mathcal{S}_{inter}\}$. Note that the different Shape Transformation methods would result in different parameters of $\mathcal{S}_{intra}$ and $\mathcal{S}_{inter}$.

Algorithm 1 Pseudocode of Slice Sharing.

Input: The rank of shared weights r and the input $\boldsymbol{x}$;
Output: The output calculated by shared weights;

```
1: # Initialize the Shared weights:
2: mdin, mdout ← Find max InFeatures and OutFeatures across all modules;
3: $\boldsymbol{A} \in \mathbb{R}^{r \times mdin}$ ← Normal Randomly Initialization;
4: $\boldsymbol{B} \in \mathbb{R}^{mdout \times r}$ ← Zero Initialization;
5:
6: # Training and Inference Stage:
7: din, dout ← InFeatures and OutFeatures of current module;
8: $\Delta \boldsymbol{W} \leftarrow \boldsymbol{B}[: dout, :]\boldsymbol{A}[:, : din]$
9: result ← $\Delta \boldsymbol{W} \boldsymbol{x}$
10:
11: return result
```

Algorithm 2 Pseudocode of Gate Transformation.

Input: The rank of shared weights r and the input $\boldsymbol{x}$;
Output: The output calculated by shared Weights;

```
1: # Initialize the Shared Weights:
2: dins, douts ← InFeatures and OutFeatures of shared weights;
3: $\boldsymbol{A}_s \in \mathbb{R}^{r \times dins}$ ← Normal Randomly Initialization;
4: $\boldsymbol{B}_s \in \mathbb{R}^{douts \times r}$ ← Zero Initialization;
5:
6: # Initialize Gate weights:
7: din, dout ← InFeatures and OutFeatures of current module;
8: $\boldsymbol{G}_{id} \in \mathbb{R}^{1 \times din}$ ← Uniform Randomly Initialization;
9: $\boldsymbol{G}_{iu} \in \mathbb{R}^{dins \times 1}$ ← Uniform Randomly Initialization;
10: $\boldsymbol{G}_{od} \in \mathbb{R}^{1 \times douts}$ ← Uniform Randomly Initialization;
11: $\boldsymbol{G}_{ou} \in \mathbb{R}^{dout \times 1}$ ← Uniform Randomly Initialization;
12:
13: # Training and Inference Stage:
14: $\boldsymbol{G}_i \in \mathbb{R}^{dins \times din} \leftarrow \boldsymbol{G}_{iu}\boldsymbol{G}_{id}$;
15: $\boldsymbol{G}_o \in \mathbb{R}^{dout \times douts} \leftarrow \boldsymbol{G}_{ou}\boldsymbol{G}_{od}$;
16: $\Delta \boldsymbol{W} \leftarrow \boldsymbol{G}_o \boldsymbol{B}_s \boldsymbol{A}_s \boldsymbol{G}_i$
17: result ← $\Delta \boldsymbol{W} \boldsymbol{x}$
18:
19: return result
```

Slice Sharing. The parameters of $\mathcal{S}$ in Slice Sharing are shared across all modules. Specifically, the parameters of intra-sharing $\mathcal{S}_{intra} = 2Ldr_{intra}$, and the parameters of inter-sharing $\mathcal{S}_{intra} = 2dr_{inter}$.

Gate Transformation. Different modules within and between layers share the parameters with the hidden dimension of d_s. Besides, each module contains an input gate and an output gate to transform the dimensions. Specifically, the parameters of intra-sharing $\mathcal{S}_{intra} = 2(Ld_s r_{intra} + M(d + d_s)))$, and the parameters of inter-sharing $\mathcal{S}_{intra} = 2(d_s r_{inter} + ML(d + d_s))$.

Kronecker Extension. Different modules within and between layers share the parameters with the hidden dimension of d_s. Besides, each module contains a Kronecker kernel $K \in \mathbb{R}^{1 \times k}$ to transform the dimensions. Specifically, the parameters of intra-sharing $\mathcal{S}_{intra} = 2L(d_s r_{intra} + Mk)$, and the parameters of inter-sharing $\mathcal{S}_{intra} = 2(d_s r_{inter} + 2MLk)$.

Algorithm 3 Pseudocode of Kronecker Extension.

Input: The rank of shared weights r and the input $\boldsymbol{x}$;
Output: The output calculated by shared weights;

```
1: # Initialize the Shared Weights:
2: dins, douts ← InFeatures and OutFeatures of shared weights;
3: $\boldsymbol{A}_s \in \mathbb{R}^{r \times dins}$ ← Normal Randomly Initialization;
4: $\boldsymbol{B}_s \in \mathbb{R}^{douts \times r}$ ← Zero Initialization;
5:
6: # Initialize Kernel weights:
7: din, dout ← InFeatures and OutFeatures of current module;
8: $k_a$ ← dins // r;
9: $k_b$ ← douts // r;
10: $\boldsymbol{K}_A \in \mathbb{R}^{1 \times k_a}$ ← Uniform Randomly Initialization;
11: $\boldsymbol{K}_B \in \mathbb{R}^{k_b \times 1}$ ← Uniform Randomly Initialization;
12:
13: # Training and Inference Stage:
14: $\boldsymbol{A} \in \mathbb{R}^{r \times d_i} \leftarrow \boldsymbol{K}_A \otimes \boldsymbol{A}_s$;
15: $\boldsymbol{B} \in \mathbb{R}^{d_o \times r} \leftarrow \boldsymbol{K}_B \otimes \boldsymbol{B}_s$;
16: $\Delta \boldsymbol{W} \leftarrow \boldsymbol{B}\boldsymbol{A}$
17: result ← $\Delta W \boldsymbol{x}$
18:
19: return result
```

A.5 Experiments

A.5.1 Results on Commonsense Reasoning

We evaluate Bi-Share LoRA for zero-shot performance on Commonsense Reasoning tasks using Llama 1-7B, Llama 1-13B, and Llama 3-8B. The results are shown in Table 2. We observe that Bi-Share LoRA consistently outperforms the baselines in terms of the average performance of Commonsense Reasoning datasets. Specifically, the Gate Transformation (GT) method of Bi-Share LoRA achieves the best average performance on the Llama 1-7B, while the Slice Sharing (SS) method achieves the best average performance on the Llama 3-8B. Furthermore, Bi-Share LoRA outperforms the baselines on 7 out of 8 datasets on Llama 1-7B, and 5 out of 8 datasets on Llama 3-8B. It is worth noting that large improvements are achieved on ARC-e, PIQA, BoolQ, and SIQA datasets. Bi-Share LoRA also achieves decent performance on the remaining datasets, including OBQA, HellaSwag, and WinG, which proves that Bi-Share LoRA is stable and reliable across different datasets. Compared to the standard LoRA with a rank of 8, Bi-Share LoRA can save about 50% trainable parameters and achieve better performance. Specifically, the Kronecker Extension (KE) method introduces fewer trainable parameters, meanwhile outperforms the baselines, and achieves on-par performance with tha SS and GT methods, indicating the more parameter-efficient method.

A.5.2 Results on MMLU Benchmark

We evaluate Bi-Share LoRA for zero-short and five-shot performance of MMLU tasks based on Llama 1-7B, Llama 1-13B, and Llama 3-8B. We demonstrate the results in Table 3. From the result, We observe that Bi-Share LoRA consistently outperforms the baselines in terms of the average performance of MMLU datasets both on zero-shot and five-shot. Specifically, the Slice Sharing (SS) method of Bi-Share LoRA achieves the best average performance on the Llama 1-7B, while the Gate Transformation (GT) method achieves the best average performance on the Llama 3-8B. Notice that, the SS method on Llama 1-7B achieves the best performance on 9 out of 10 metrics and achieves the second-best performance for the remaining one. Furthermore, the GT and Kronecker Extension (KE) Bi-Share LoRA achieve the second-best performance than SS and consistently outperform the baselines. Compared to the standard LoRA with a rank of 8, Bi-Share LoRA can save about 50% trainable parameters and achieve better performance. Specifically, the KE method outperforms the baselines and achieves on-par performance with the SS and GT methods, indicating the most parameter-efficient method.

A.6 Experiments On QWen model

To further demonstrate the effectiveness of Bi-Share LoRA, we also conduct experiments on QWen 2.5-7B (Yang et al., 2024)with the specific versions detailed in Appendix A.12. We fine-tune the QWen 2.5-7B model on the Alpaca dataset, and evaluation its performance on the Common Reasoning task datasets, the experiment results are shown in Table 9.

Table 9: Results of Zero-shot performance on Qwen 2.5-7B, in Bi-Share LoRA and baselines on the Commonsense Reasoning benchmark. We report the number of trainable parameters (# params) and the corresponding ratio for each method.

	Methods	# params	ratio	OBQA	ARC-c	HellaSwag	ARC-e	PIQA	WinoG.	BoolQ	SIQA	Avg.
QWen 2.5-7B	$\text{LoRA}_{r=8}$	14.16M	0.18%	44.00	47.27	71.34	76.68	77.09	67.48	84.83	49.54	64.78
	$\text{VeRA}_{r=64}$	0.80M	0.01%	**48.20**	51.45	79.54	79.59	78.45	68.82	85.90	50.36	67.79
	$\text{VB-LoRA}_{r=4}$	2.51M	0.03%	**48.20**	52.30	**79.57**	79.38	78.45	**70.24**	85.69	50.56	68.05
	Bi-Share-LoRA (SS)	7.67M	0.10%	45.60	**55.55**	79.09	**80.98**	79.16	70.17	**86.39**	**51.02**	**68.49**
	Bi-Share-LoRA (GT)	8.03M	0.10%	45.00	51.45	73.96	80.22	78.18	**70.24**	85.47	50.67	66.90
	Bi-Share-LoRA (KE)	3.83M	0.05%	47.00	53.67	77.68	79.88	**79.82**	69.85	86.18	50.15	68.03

A.7 Experiments On FLAN_V2 dataset

To further demonstrate the effectiveness of Bi-Share LoRA, we also conduct experiments on Llama 3-8B on the FLAN_V2 instruction dataset Chung et al. (2022) [2], which is an another dataset for instruction tuning. We conduct eperiments on the Chain Of Thought task and evaluation its performance on the Common Reasoning task datasets, the experiment results are shown in Table 10.

Table 10: Results of Zero-shot performance on Llama 3-8B, in Bi-Share LoRA and baselines on Commonsense Reasoning benchmark. We report the number of trainable parameters (# params) and the corresponding ratio for each method.

	Methods	# params	ratio	OBQA	ARC-c	HellaSwag	ARC-e	PIQA	WinoG.	BoolQ	SIQA	Avg.
Llama 3-8B	$\text{LoRA}_{r=8}$	14.16M	0.18%	45.40	53.41	79.20	80.43	79.82	74.35	83.27	47.39	67.91
	$\text{VeRA}_{r=64}$	0.80M	0.01%	44.80	53.92	79.14	79.92	79.54	72.69	80.95	47.03	67.25
	$\text{VB-LoRA}_{r=4}$	2.51M	0.03%	44.00	54.01	78.72	80.26	78.62	74.66	81.19	46.01	67.18
	Bi-Share-LoRA (SS)	7.67M	0.10%	**46.60**	**56.23**	79.00	81.14	80.79	74.27	83.52	48.77	68.79
	Bi-Share-LoRA (GT)	8.03M	0.10%	45.20	55.38	**79.22**	81.99	**81.28**	**74.98**	83.21	48.52	68.72
	Bi-Share-LoRA (KE)	3.83M	0.05%	44.40	55.03	79.11	**82.41**	81.18	74.90	**84.10**	**49.54**	**68.83**

A.8 Analysis

A.9 Contribution Analysis

We further investigate how performance is affected when intra-layer and inter-layer parameter sharing are applied independently under different rank settings. Using the LLaMA 1-7B model fine-tuned on the Alpaca dataset, we configured various shared rank settings for different shape transformation modalities and evaluated the results on the CommonSense task. The findings, presented in Tabke 11, demonstrate that intra-layer sharing alone outperforms inter-layer sharing alone in terms of performance. However, inter-layer sharing is significantly more parameter-efficient, achieving comparable results with fewer parameters. This trade-off underscores the complementary benefits of combining both strategies.

Additionally, we observed that using the GT approach to increase the rank for either intra-layer or inter-layer sharing alone does not lead to significant performance improvements and may even degrade performance. In contrast, for the SS and KE methods, increasing the rank consistently enhances model performance, further emphasizing the benefits of tailored sharing strategies.

A.10 Rank Allocation

We designed comparative experiments to explore the impact of adjusting different rank ratios on fine-tuning performance. Specifically, we varied the rank value of one sub-LoRA module while

[2] `https://huggingface.co/datasets/BEE-spoke-data/flan-v2-hf`

Table 11: Different configurations of ranks for our Bi-Share LoRA. **Boldface** denotes the best results in terms of the corresponding metrics, and underline means the second-best performance.

Method	Mode	Ranks	OBQA	ARC-c	HellaSwag	ARC-e	PIQA	WinoG.	BoolQ	SIQA	Avg.
Adjust intra	SS	0,8,0	45.00	**47.70**	**77.32**	76.64	**80.25**	70.01	78.10	48.41	65.43
	SS	0,16,0	45.20	47.10	77.31	**77.02**	80.14	**70.09**	78.01	48.82	65.46
	SS	0,32,0	**45.40**	47.35	77.18	76.47	80.09	69.69	**78.50**	**49.18**	**65.48**
	GT	0,8,0	43.20	**45.99**	**76.52**	**75.63**	79.11	69.85	**74.98**	**46.16**	**63.93**
	GT	0,16,0	**43.80**	45.48	75.71	75.55	**79.38**	69.53	74.16	45.70	63.66
	GT	0,32,0	43.60	44.88	76.03	75.34	79.00	**70.01**	73.00	46.11	63.50
	KE	0,8,0	45.40	**48.04**	77.16	76.35	**80.20**	**70.72**	75.90	46.78	65.07
	KE	0,16,0	**45.80**	46.93	77.48	**76.73**	80.03	70.56	75.93	**47.44**	**65.11**
	KE	0,32,0	45.20	47.10	**77.53**	76.39	79.82	70.24	**75.96**	47.13	64.92
Adjust inter	SS	0,0,8	**44.60**	48.29	77.23	**77.02**	**80.20**	69.85	77.74	**48.31**	65.40
	SS	0,0,16	44.40	**48.72**	77.35	76.64	80.09	70.32	78.23	47.95	65.46
	SS	0,0,32	44.40	47.78	**77.36**	76.94	80.03	**70.80**	**78.75**	48.21	**65.53**
	GT	0,0,8	**44.40**	45.56	76.41	75.67	**79.38**	**69.85**	**75.84**	45.75	64.11
	GT	0,0,16	44.00	**46.33**	**76.87**	**76.14**	79.11	69.30	74.40	**46.93**	**64.13**
	GT	0,0,32	**44.40**	44.97	76.00	72.73	79.11	69.38	75.11	45.75	63.43
	KE	0,0,8	43.80	45.65	75.70	75.63	**79.38**	70.01	75.66	46.47	64.04
	KE	0,0,16	45.80	**46.93**	**76.59**	**76.77**	79.16	**70.64**	76.57	47.75	**65.03**
	KE	0,0,32	**46.00**	45.56	76.17	76.30	79.16	70.01	**76.79**	**47.80**	64.72

keeping the ranks of the other two fixed. We conducted experiments on the rank configurations for local, intra-sharing, and inter-sharing. The results are shown in Table 12. Assigning a lower rank to the local component and a higher rank to the shared parameters yielded better performance, further illustrating the redundancy in the standard LoRA parameters.

Table 12: Different configuration of ranks for our Bi-Share LoRA (KE). **Boldface** denotes the best results in terms of the corresponding metrics, and underline means the second-best performance.

Method	Ranks	OBQA	ARC-c	HellaSwag	ARC-e	PIQA	WinoG.	BoolQ	SIQA	Avg.
Adjust local	2,4,16	**45.00**	**47.78**	**77.45**	76.85	79.92	**69.69**	78.07	48.57	65.42
	4,4,16	44.80	47.61	77.35	77.02	**80.30**	69.50	77.74	48.31	65.33
	8,4,16	44.80	47.70	77.33	**77.19**	**80.30**	69.53	77.80	**48.82**	**65.43**
	16,4,16	44.60	47.18	77.38	76.68	80.20	**69.69**	**78.16**	48.52	65.30
Adjust intra	2,2,16	**45.40**	46.93	77.43	76.56	80.14	69.77	78.20	**48.93**	**65.42**
	2,4,16	45.00	**47.78**	**77.45**	76.85	79.92	69.69	78.07	48.57	**65.42**
	2,8,16	44.80	47.44	77.36	**77.06**	79.43	**70.24**	77.43	48.52	65.29
	2,16,16	45.20	46.84	77.38	76.52	**80.41**	69.61	**78.38**	48.77	65.39
Adjust inter	2,4,8	**45.40**	47.01	77.32	76.73	80.09	70.17	**78.17**	48.41	65.41
	2,4,16	45.00	**47.78**	**77.45**	76.85	79.92	69.69	78.07	**48.57**	**65.42**
	2,4,32	44.20	47.35	77.28	**76.94**	79.92	**70.48**	77.86	48.31	65.29
	2,4,64	44.80	46.67	77.31	76.89	**80.30**	69.93	77.77	48.36	65.26

A.11 MULTI-LORA SERVING ANALYSIS.

In a multi-LoRA deployment, tasks share a common pre-trained model, adapted to each task by loading specific LoRA parameters. Typically, all LoRA parameters are preloaded into GPU memory to minimize task-switching latency and maximize GPU utilization. However, with many tasks, only frequently used parameters remain in GPU memory, while others are stored on the CPU, causing delays during frequent task switching. To mitigate this, we reduce the parameter size through sharing, allowing more parameters to fit in GPU memory.

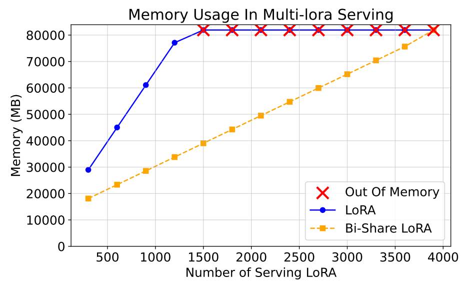

Figure 6: Memory usage comparison of LoRA and Bi-Share LoRA (GE) for serving different numbers of tasks.

We conducted a comparative experiment to analyze the GPU memory footprint of deploying different numbers of downstream tasks using the Llama model. Specifically, we deployed a Llama 1-7B base model on an A100 80G GPU and measured the memory usage for 100 to 1500 LoRA parameters and Bi-Share LoRA (GE) parameters during inference. The results, shown in Fig 6, indicate that Bi-Share LoRA reduces memory usage by approximately 60% compared to standard LoRA under the same number of parameters. Furthermore, when deploying more than 1200 LoRA parameters, the memory usage exceeds the GPU's capacity. Furthermore, to determine the maximum deployment capacity, we incrementally added parameters until the memory limit was reached. Our findings reveal that standard LoRA can load up to 1207 parameters, while Bi-Share LoRA supports up to 3878, significantly enhancing deployment efficiency.

A.12 LLM versions.

We provide the Hugging Face link of LLMs used in the experiment: Llama 1-7B: `https://huggingface.co/baffo32/decapoda-research-llama-7B-hf`; Llama 3-8B: `https://huggingface.co/meta-llama/Meta-Llama-3.1-8B`. Llama 1-13B: `https://huggingface.co/yahma/llama-13b-hf`. Qwen2.5-7B: `https://huggingface.co/Qwen/Qwen2.5-7B-Instruct`

A.13 Software and hardware configuration.

Our implementation utilizes the following configurations: *PyTorch* version 2.1.2, *Transformers* library version 4.41.0, *PEFT (Parameter-Efficient Fine-Tuning)* library version 0.11.1, *CUDA* version 12.4, *GPU:* NVIDIA V100 GPU with 32GB of memory, NVIDIA A100 GPU with 80GB, *Operating System:* Ubuntu.

A.14 Datasets and Benchmarks

BoolQ (Clark et al., 2019) is a dataset for yes/no question answering. It consists of naturally occurring questions paired with passages extracted from Wikipedia. It is part of the SuperGLUE benchmark, a suite of challenging NLP tasks. It is used to assess a model's ability to perform reading comprehension and binary classification based on the context provided.

PIQA (Bisk et al., 2020) is a dataset designed to evaluate commonsense reasoning about physical interactions. It contains multiple-choice questions related to everyday physical tasks, asking models to choose the most plausible way of completing or describing an action. It is used as a benchmark for evaluating the commonsense reasoning abilities of language models, particularly in the context of tasks requiring physical understanding.

HellaSwag (Zellers et al., 2019) is a large-scale dataset for evaluating commonsense reasoning and natural language inference. The task involves selecting the most plausible continuation of a given story or event description from multiple choices. It is used to benchmark models on their ability to perform commonsense reasoning, particularly in cases where the correct answer requires understanding context, sequencing, and implications.

WinoGrande (Sakaguchi et al., 2021) is a large-scale dataset for commonsense reasoning, specifically designed to address the limitations of the Winograd Schema Challenge. The task involves resolving pronoun references in sentences, where the correct interpretation requires commonsense knowledge. It is used as a benchmark for evaluating models on their ability to perform pronoun resolution and commonsense reasoning.

ARC-easy (Clark et al., 2018) and ARC-challenge (Clark et al., 2018) are part of the AI2 Reasoning Challenge, designed to evaluate a model's ability to answer multiple-choice questions that require complex reasoning and background knowledge. They are used as a benchmark for testing advanced question-answering systems, especially those requiring sophisticated reasoning, knowledge integration, and inference capabilities.

OpenbookQA (Mihaylov et al., 2018) is a multiple-choice question-answering dataset that focuses on elementary science questions. The dataset comes with an "open book" of scientific facts, and models must combine this knowledge with reasoning to answer the questions correctly. It is used as a benchmark for evaluating a model's ability to perform open-domain question answering, where success requires not just knowledge retrieval but also reasoning and application of that knowledge.

Social QA (Sap et al., 2019) , often abbreviated as SIQA, is composed of question-answer pairs that simulate real-world information-seeking dialogues. This dataset is designed to assess the capability of models to engage in information-seeking conversations, where the model must ask clarifying questions to a human user to gather information and then provide an answer to the original query.

MMLU (Hendrycks et al., 2021) is a benchmark designed to assess a model's world knowledge and problem-solving abilities across a wide range of subjects. It evaluates models in both zero-shot and few-shot settings, making the tasks more challenging and aligned with human evaluation methods. The benchmark spans 57 subjects, including STEM, humanities, social sciences, and other fields, with difficulty levels ranging from elementary to advanced professional. Each question presents four answer choices, and the task is to predict the correct one based on the given instruction.