SHARE ONCE, ADAPT EVERYWHERE: HIGH-RANK ADAPTATION ACROSS LAYERS AND MODULES IN LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent high-rank parameter-efficient fine-tuning (PEFT) methods improve the expressiveness of weight updates while retaining a separate set of trainable parameters for each target weight matrix. This leaves open whether high-rank adaptation can accommodate more extensive parameter sharing across layers and modules. We introduce SHARE, a high-rank adaptation method that organizes target weight matrices according to their update shapes and reuses a small set of trainable parameter banks across layers and modules. Weight matrices assigned to the same bank reuse the corresponding high-rank update. We provide a first-order analysis showing that fixed parameter sharing defines a reparameterization of the weight updates and derive the resulting update to the model weights. The shared structure also allows each high-rank update to be constructed once and reused by all target weight matrices assigned to the same bank during training. Extensive experiments on commonsense reasoning, mathematical reasoning, and code generation across multiple backbone models show that SHARE consistently outperforms strong PEFT baselines, while achieving competitive training throughput.

1 INTRODUCTION

In recent years, large language models (LLMs) (Brown et al., 2020; Chen et al., 2021; Zhang et al., 2022; Touvron et al., 2023) have achieved remarkable progress in natural language understanding, generation, and reasoning. Despite their strong general-purpose capabilities, these models often require further fine-tuning for specific downstream tasks or domains. However, full fine-tuning updates all model parameters and therefore incurs substantial memory and computational costs for modern LLMs. Parameter-efficient fine-tuning (PEFT) methods (Houlsby et al., 2019; Lester et al., 2021; Hu et al., 2022; Liu et al., 2024; Wang et al., 2025) address this issue by keeping the pretrained weights frozen and optimizing only a small number of task-specific parameters.

Among existing PEFT methods, low-rank adaptation (LoRA) (Hu et al., 2022) has become one of the most widely adopted approaches because of its simplicity and ease of deployment. LoRA represents the update to a frozen weight matrix using two low-rank matrices, substantially reducing the number of trainable parameters, while allowing the learned update to be merged into the pretrained model after training. However, restricting the update itself to low rank can limit its flexibility, particularly for tasks that require more complex changes to the pretrained model (Liu et al., 2023; Jiang et al., 2024; Huang et al., 2025).

Recent studies have therefore explored more expressive high-rank adaptation under limited parameter budgets. HiRA (Huang et al., 2025) constructs high-rank updates through a Hadamard-product parameterization, while MoSA (Wang et al., 2026) shares trainable parameters among groups of weight entries to produce fine-grained high-rank updates. These methods show that a small number of trainable parameters can still produce high-rank weight updates. However, they generally assign a separate set of trainable parameters to each target weight matrix. Although low-rank methods such as VeRA (Kopiczko et al., 2024) and ShareLoRA (Song et al., 2024) have explored parameter sharing across layers, such sharing has received much less attention in high-rank adaptation. Therefore, we ask whether the same trainable parameters can be reused across layers and modules in high-rank adaptation.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To answer this question, we introduce Shared High-Rank Adaptation REuse (SHARE). SHARE is a high-rank adaptation method that reuses a small number of shared trainable parameter sets, which we refer to as parameter banks, across layers and modules. Target weight matrices are assigned to these banks according to their update shapes, and matrices assigned to the same bank reuse the same high-rank update. Unlike low-rank sharing methods that share components used to construct separate adapters, SHARE directly shares the high-rank update across multiple weight matrices. As a result, the same trainable parameters can be reused throughout the model without maintaining a separate set for every target weight matrix. We further show from a first-order perspective that parameter sharing can be written as a fixed mapping from the shared parameters to the model weights, and derive the resulting weight update. The analysis applies to different fixed sharing patterns and does not depend on a particular relationship between the layers or modules that share parameters. This gives us flexibility in how the parameters are shared. We therefore group weight matrices by update shape, which allows the same high-rank update to be constructed once and reused by multiple matrices. When multiple matrices share the same update and receive the same input, the corresponding matrix multiplication can be reused as well.

We evaluate SHARE on commonsense reasoning, mathematical reasoning, and code generation across multiple backbone models. Under matched parameter budgets, SHARE consistently achieves strong task performance while maintaining competitive training throughput. These results show that high-rank adaptation does not require a separate set of trainable parameters for every target weight matrix, and that extensive parameter sharing can support both effective and efficient model adaptation.

In summary, the contributions of this paper are as follows.

- We propose SHARE, a high-rank adaptation method that reuses a small number of trainable parameter banks across layers and modules.
- We provide a first-order analysis of how shared parameters determine the resulting weight updates, together with an efficient implementation that reduces redundant computation during training.
- Extensive experiments across multiple backbone models and downstream tasks demonstrate the effectiveness of SHARE.

2 RELATED WORKS

PEFT and Low-Rank Adaptation. PEFT adapts pretrained models by optimizing only a small number of parameters. Representative approaches include prompt-based methods (Lester et al., 2021), adapters (Houlsby et al., 2019), and methods that directly parameterize weight updates (Aghajanyan et al., 2021; Guo et al., 2021; Hu et al., 2022). Among them, LoRA (Hu et al., 2022) represents the weight update as $\Delta W = BA$ using two low-rank factors, substantially reducing the number of trainable parameters while allowing the update to be merged into the pretrained weights after training. Its effectiveness has inspired many extensions to the low-rank parameterization. AdaLoRA (Zhang et al., 2023) dynamically allocates the rank budget across weight matrices, DoRA (Liu et al., 2024) decomposes the weight into magnitude and direction, and LoRA-Pro (Wang et al., 2025) modifies the optimization of the low-rank factors to better approximate full fine-tuning. Despite their different motivations, these methods retain low-rank factors as a central part of the adaptation.

Parameter Sharing in PEFT. Parameter sharing has also been explored to improve the efficiency of low-rank adaptation. VeRA (Kopiczko et al., 2024) shares low-rank matrices across layers and learns lightweight scaling vectors, while ShareLoRA (Song et al., 2024) shares LoRA factors across layers to reduce parameter and memory costs. These methods mainly use sharing to reduce the number of parameters or the memory required by low-rank adaptation, leaving parameter sharing in high-rank adaptation less explored.

High-Rank Adaptation. Recent work has relaxed the low-rank constraint of LoRA. MoRA (Jiang et al., 2024) uses a trainable square matrix with fixed transformations to produce higher-rank updates, while HiRA (Huang et al., 2025) constructs $\Delta W = W_0 \odot (AB)$, whose rank can exceed that of AB. MoSA (Wang et al., 2026) directly groups weight entries and assigns one trainable parameter to each group, avoiding matrix factorization. These methods show that parameter-efficient adaptation does

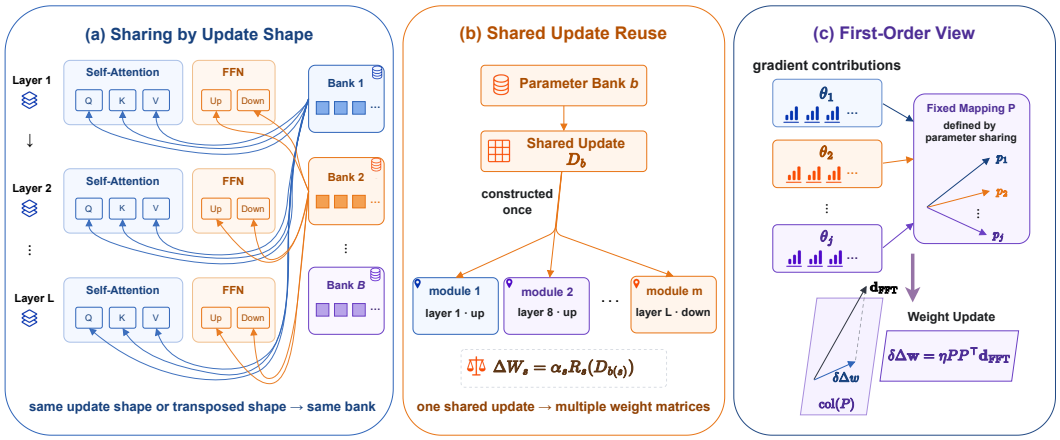

Figure 1: Overview of SHARE. (a) Target weight matrices are assigned to parameter banks according to their update shapes, allowing the same bank to be reused across layers and modules. (b) For each bank, the shared update is constructed once during a forward pass and reused by multiple weight matrices. (c) From a first-order perspective, parameter sharing defines a fixed mapping from the shared parameters to the weight updates, which determines the resulting weight-space update.

not require the resulting weight update to remain low-rank. However, they still generally assign a separate set of trainable parameters to each target weight matrix. SHARE instead reuses a small number of trainable parameter banks across layers and modules while retaining high-rank weight updates.

3 METHODOLOGY

In this section, we present SHARE, a high-rank adaptation method that reuses a small number of trainable parameter banks across layers and modules. Target weight matrices are grouped by update shape, with transposed shapes sharing the same bank through transposition. Within each bank, a fixed assignment maps update entries to trainable parameters, directly constructing the update in the full matrix space. Figure 1 illustrates the overall design. We first introduce the shared parameterization, then analyze its first-order behavior, and finally describe the efficient implementation.

3.1 SHARED PARAMETERIZATION

Problem Setup. Let $\mathcal{S}$ denote the set of weight matrices selected for adaptation. For each $s \in \mathcal{S}$, let $W_s \in \mathbb{R}^{d_s^{\text{out}} \times d_s^{\text{in}}}$ denote a frozen pretrained weight matrix. Given an input X_s, the corresponding linear transformation is

$$Y_s = X_s(W_s + \Delta W_s)^\top, \tag{1}$$

where $\Delta W_s \in \mathbb{R}^{d_s^{\text{out}} \times d_s^{\text{in}}}$ denotes the weight update for W_s. Existing high-rank PEFT methods generally parameterize each ΔW_s separately. In contrast, SHARE constructs the updates of multiple weight matrices from the same trainable parameter bank.

Grouping Weight Matrices via Update Shapes. As illustrated in Figure 1(a), we organize the target weight matrices according to the shapes of their updates. Weight matrices with the same update shape share the same parameter bank, while matrices with transposed update shapes can reuse the same bank through matrix transposition. Specifically, we partition $\mathcal{S}$ into disjoint subsets $\{\mathcal{S}_b\}_{b=1}^{B}$, where B denotes the number of parameter banks. Let $b(s) \in \{1, \ldots, B\}$ denote the index of the bank assigned to s. Each bank b provides a shared update matrix $D_b \in \mathbb{R}^{m_b \times n_b}$. For every $s \in \mathcal{S}_b$,

the shape of W_s is either $m_b \times n_b$ or $n_b \times m_b$. We define an operator $\mathcal{R}_s : \mathbb{R}^{m_b \times n_b} \to \mathbb{R}^{d_s^{\text{out}} \times d_s^{\text{in}}}$ as

$$\mathcal{R}_s(D) = \begin{cases} D, & \text{if } (d_s^{\text{out}}, d_s^{\text{in}}) = (m_b, n_b), \\ D^\top, & \text{if } (d_s^{\text{out}}, d_s^{\text{in}}) = (n_b, m_b). \end{cases} \tag{2}$$

The number of banks therefore depends on the number of different shapes of the selected weight matrices. For example, the query, key, and value projections in the attention module can share one bank when their shapes are the same. Under grouped-query attention, the query projection has a different shape from the key and value projections and so it uses a separate bank. The up and down projections in the feed-forward module can likewise share one bank when their update shapes are transposes of each other.

Constructing the Bank Update. For each bank b with matrix shape $m_b \times n_b$, let K_b denote its parameter budget. We maintain the trainable parameter vector

$$\boldsymbol{\theta}_b = [\theta_{b,1}, \ldots, \theta_{b,K_b}]^\top \in \mathbb{R}^{K_b}. \tag{3}$$

An index mapping function $\kappa_b(i, j) \in \{1, \ldots, K_b\}$ assigns each entry of the update matrix to a specific trainable parameter. The shared update D_b is then constructed as

$$D_b[i, j] = \theta_{b,\kappa_b(i,j)}. \tag{4}$$

The assignment is generated once before training and remains fixed throughout training. Because D_b is constructed directly as a matrix rather than through a low-rank factorization, its rank is not restricted to a predefined low rank.

Applying a Shared Update to Weight Matrices. For each $s \in \mathcal{S}_b$, the update is obtained from the shared update D_b as

$$\Delta W_s = \alpha_s \mathcal{R}_s(D_{b(s)}), \tag{5}$$

where $\alpha_s \in \mathbb{R}$ is a fixed coefficient associated with W_s. Thus, all weight matrices in $\mathcal{S}_b$ share the same trainable parameters $\boldsymbol{\theta}_b$, while their updates may differ through transposition and the coefficient α_s. The coefficients remain fixed throughout training and introduce no additional trainable parameters. In the default configuration, they are sampled using random signs with small continuous magnitude perturbations with details provided in Appendix A.3.

3.2 First-Order Analysis of Parameter Sharing

We analyze the first-order effect of parameter sharing on weight updates, first for a single shared parameter and then for the full SHARE parameterization. Figure 1(c) illustrates this view.

A Single Shared Parameter. Consider a trainable parameter $\theta \in \mathbb{R}$ shared by c weight entries. Let $\Delta\mathbf{u} = [\Delta u_1, \ldots, \Delta u_c]^\top \in \mathbb{R}^c$ denote the vector of updates to these c weight entries. Let $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_c]^\top \in \mathbb{R}^c$ contain the fixed coefficients that determine how θ contributes to these entries, so that $\Delta\mathbf{u} = \theta\boldsymbol{\alpha}$. Let $\mathbf{g}_u = \nabla_{\Delta\mathbf{u}}\mathcal{L}$ denote the gradient with respect to $\Delta\mathbf{u}$. By the chain rule, the gradient of $\mathcal{L}$ with respect to θ is

$$\nabla_\theta \mathcal{L} = \sum_{i=1}^{c} \alpha_i g_{u,i} = \boldsymbol{\alpha}^\top \mathbf{g}_u, \tag{6}$$

where $g_{u,i}$ denotes the i-th entry of $\mathbf{g}_u$. Thus, the gradient of a shared parameter combines the gradient contributions from all weight entries that it controls. A gradient-descent step $\delta\theta = -\eta\boldsymbol{\alpha}^\top\mathbf{g}_u$ produces $\delta\Delta\mathbf{u} = -\eta(\boldsymbol{\alpha}^\top\mathbf{g}_u)\boldsymbol{\alpha}$, which lies along the fixed direction $\boldsymbol{\alpha}$. Gradient summation is a special case of Eq. (6). Specifically, when the shared trainable parameter θ contributes equally to all weight entries, i.e., $\boldsymbol{\alpha} = \mathbf{1}_c$, Eq. (6) reduces to $\nabla_\theta\mathcal{L} = \sum_{i=1}^{c} g_{u,i}$.

Extension to SHARE. We now extend the same view to the full parameterization of SHARE. For bank b, let $H_b \in \{0, 1\}^{m_b n_b \times K_b}$ be the fixed assignment matrix induced by κ_b in Eq. (4). Each row of H_b contains exactly one nonzero entry, indicating the trainable parameter assigned to the corresponding entry of D_b. It follows that $\text{vec}(D_b) = H_b\boldsymbol{\theta}_b$, where $\text{vec}(\cdot)$ denotes the vectorization of

a matrix by stacking its columns. For each $s \in \mathcal{S}_b$, let R_s denote the permutation matrix corresponding to $\mathcal{R}_s$ in vectorized form. The update to W_s can then be written as $\mathrm{vec}(\Delta W_s) = \alpha_s R_s H_b \boldsymbol{\theta}_b$.

Let $\Delta\mathbf{w}$ denote the vector obtained by stacking $\mathrm{vec}(\Delta W_s)$ for all $s \in \mathcal{S}$. Similarly, let $\boldsymbol{\theta} = [\boldsymbol{\theta}_1^\top, \ldots, \boldsymbol{\theta}_B^\top]^\top$ denote the vector containing all trainable bank parameters. The resulting parameterization can then be written as

$$\Delta\mathbf{w} = P\boldsymbol{\theta}, \tag{7}$$

where P is the fixed matrix determined by the assignments H_b, the transformations R_s, and the coefficients α_s. Parameter sharing in SHARE therefore defines a fixed linear mapping from the trainable parameters to the weight updates. The same formulation also allows more extensive sharing than the default grouping by update shape. In particular, a single parameter bank can be shared across weight matrices with different update shapes through separate fixed assignments. We evaluate this variant in Appendix B.2.

Weight Updates under Parameter Sharing. Let $\mathbf{g} = \nabla_{\Delta\mathbf{w}}\mathcal{L}$ denote the gradient with respect to the selected weight updates. Let θ_j denote the j-th entry of $\boldsymbol{\theta}$, and let $\mathbf{p}_j$ denote the corresponding j-th column of P. We refer to $\mathbf{p}_j$ as the projection axis associated with θ_j, since changing θ_j moves $\Delta\mathbf{w}$ along the direction $\mathbf{p}_j$.

In SHARE, different trainable parameters are assigned to disjoint weight entries. The corresponding projection axes therefore have disjoint support and are mutually orthogonal. By construction, every trainable parameter is assigned to at least one weight entry, so each column of P is nonzero. Hence, P has full column rank and $P^\top P$ is diagonal and invertible.

The column space of P, denoted by $\mathrm{col}(P)$, is the subspace spanned by the projection axes and represents the weight-update directions reachable through the shared parameterization. The orthogonal projection of $\mathbf{g}$ onto this subspace is

$$\Pi_P\mathbf{g} = P(P^\top P)^{-1}P^\top\mathbf{g}. \tag{8}$$

Theorem 1 (First-Order Alignment under Parameter Sharing) *Consider the linear parameterization $\Delta\mathbf{w} = P\boldsymbol{\theta}$. Let $\mathbf{d}_{\mathrm{FFT}} = -\mathbf{g}$ denote the full fine-tuning descent direction restricted to the selected weight entries. Under gradient descent with learning rate $\eta > 0$, if $P^\top\mathbf{g} \neq \mathbf{0}$, the induced update $\delta\Delta\mathbf{w}$ forms an acute angle with $\mathbf{d}_{\mathrm{FFT}}$. Furthermore, if all projection axes have the same squared ℓ_2 norm $\gamma > 0$, i.e., $P^\top P = \gamma I$, the induced update is proportional to the orthogonal projection of $\mathbf{d}_{\mathrm{FFT}}$ onto $\mathrm{col}(P)$.*

Proof. By the chain rule, $\nabla_{\boldsymbol{\theta}}\mathcal{L} = P^\top\mathbf{g}$, so a gradient-descent step gives $\delta\boldsymbol{\theta} = -\eta P^\top\mathbf{g}$. Since $\Delta\mathbf{w} = P\boldsymbol{\theta}$, the corresponding weight-space update is

$$\delta\Delta\mathbf{w} = P\delta\boldsymbol{\theta} = -\eta PP^\top\mathbf{g}. \tag{9}$$

Using $\mathbf{d}_{\mathrm{FFT}} = -\mathbf{g}$,

$$\mathbf{d}_{\mathrm{FFT}}^\top\delta\Delta\mathbf{w} = \eta\,\mathbf{g}^\top PP^\top\mathbf{g} = \eta\|P^\top\mathbf{g}\|_2^2 > 0, \tag{10}$$

whenever $P^\top\mathbf{g} \neq \mathbf{0}$. Hence, $\delta\Delta\mathbf{w}$ and $\mathbf{d}_{\mathrm{FFT}}$ form an acute angle.

If $P^\top P = \gamma I$ with $\gamma > 0$, then Eq. (8) gives $\Pi_P = \gamma^{-1}PP^\top$. Therefore,

$$\delta\Delta\mathbf{w} = -\eta\gamma\,\Pi_P\mathbf{g} = \eta\gamma\,\Pi_P\mathbf{d}_{\mathrm{FFT}}. \tag{11}$$

This shows that the induced update is proportional to the projected full fine-tuning direction. □

The proof also shows that the j-th component of $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is $\mathbf{p}_j^\top\mathbf{g}$. Since a single trainable parameter may control weight entries across multiple layers or modules, this quantity combines the corresponding gradient contributions, extending the single-parameter interpretation in Eq. (6) to the full parameterization. More generally, when the projection axes have different norms, the update remains in $\mathrm{col}(P)$ but its components along different axes receive different scaling factors. The analysis above assumes standard gradient descent. With an adaptive optimizer such as AdamW, the update to $\boldsymbol{\theta}$ is modified by the optimizer, but $\delta\Delta\mathbf{w} = P\delta\boldsymbol{\theta}$ still lies in $\mathrm{col}(P)$. Appendix B.1 further compares this parameterization with LoRA.

3.3 Efficient Implementation

Caching Shared Updates. As illustrated in Figure 1(b), the same shared update D_b is reused by all weight matrices assigned to bank b. A direct implementation would reconstruct D_b each time the bank is used by another weight matrix, but this introduces redundant computation and unnecessary memory-access overhead. Instead, we construct D_b once for each bank during a forward pass and cache the resulting tensor, applying transposition when required by $\mathcal{R}_s$. Because the cached tensor remains connected to the same trainable parameters $\boldsymbol{\theta}_b$, gradients from all weight matrices that use the bank are accumulated correctly during backpropagation. This avoids repeatedly constructing the same high-rank update across layers and modules.

Reusing Matrix Multiplications. The computational cost can be reduced further when multiple weight matrices not only use the same shared update but also receive the same input. Suppose that several matrices in $\mathcal{S}_b$ receive the same input X and use D_b without transposition. In this case, we compute $Z_b = XD_b^\top$ once, and the contribution of each matrix s is obtained as $\Delta Y_s = \alpha_s Z_b$. In standard multi-head attention, the query, key, and value weight matrices can therefore reuse a single matrix multiplication when they share the same bank. Under grouped-query attention, the query weight matrix uses a separate bank because of its different shape, while the key and value weight matrices can reuse the same matrix multiplication. Weight matrices that share a bank but receive different inputs, or require different forms of D_b through transposition, still reuse the cached shared update but require separate matrix multiplications.

4 Experiments

We evaluate SHARE on commonsense reasoning, mathematical reasoning, and code generation across three backbone models: Llama-2-7B (Touvron et al., 2023), Llama-3-8B (Grattafiori et al., 2024), and Qwen3-4B (Team et al., 2025). We compare its task performance and training efficiency with representative low-rank and high-rank PEFT methods. We further provide ablation studies and analyses to better understand the behavior of SHARE.

4.1 Experimental Setup

Tasks and Datasets. We consider three types of downstream tasks. For commonsense reasoning, we train on BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), OpenBookQA (Mihaylov et al., 2018), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2021), ARC-Challenge, and ARC-Easy (Clark et al., 2018), and evaluate performance on each dataset. For mathematical reasoning, we train on a subset of 100K examples from MetaMathQA (Yu et al., 2024) and evaluate on GSM8K (Cobbe et al., 2021). For code generation, we train on a subset of 100K examples from CodeFeedback (Zheng et al., 2024) and evaluate on HumanEval (Chen et al., 2021).

Backbone Models and Baselines. We conduct experiments with Llama-2-7B, Llama-3-8B, and Qwen3-4B. We compare SHARE with LoRA (Hu et al., 2022), DoRA (Liu et al., 2024), HiRA (Huang et al., 2025), and MoSA (Wang et al., 2026). LoRA and DoRA retain low-rank factors in their parameterization, while HiRA and MoSA allow the resulting weight updates to have higher rank. For each backbone, we configure all methods under matched parameter budgets, using LoRA with rank $r = 32$ for Llama-2-7B and Llama-3-8B and rank $r = 16$ for Qwen3-4B as the reference budget.

Training Details. We adapt the query, key, and value weight matrices in the attention modules and the up and down weight matrices in the feed-forward modules. All models are trained for three epochs with a global batch size of 48. We use the AdamW optimizer with bfloat16 precision, a cosine learning-rate schedule, and a warmup ratio of 0.1. The learning rate is set to 1×10^{-5} for commonsense reasoning and 2×10^{-5} for mathematical reasoning and code generation.

Evaluation Metrics. For commonsense reasoning, we report accuracy on each of the eight datasets together with their unweighted average. For GSM8K, we report accuracy based on the final numerical answer, and for HumanEval, we report Pass@1. All experiments are conducted with three random seeds, with the mean reported for all experiments and the standard deviation additionally reported for the main results.

Table 1: Commonsense reasoning accuracy (%). Each entry reports the mean ± standard deviation over three random seeds. The best mean is shown in **bold**, and the second-best mean is underlined.

Method	BoolQ	PIQA	SIQA	ARC-C	ARC-E	OBQA	HellaSwag	WinoG	Avg.
Llama-2-7B ($r = 32$)									
LoRA	66.15 ± 0.58	82.57 ± 0.49	79.93 ± 0.46	73.08 ± 0.55	86.47 ± 0.38	79.99 ± 0.76	93.35 ± 0.23	84.31 ± 0.65	80.73 ± 0.47
DoRA	66.94 ± 0.74	82.69 ± 0.28	79.96 ± 0.44	72.92 ± 0.82	86.60 ± 0.67	80.55 ± 0.61	93.45 ± 0.11	84.88 ± 0.27	81.00 ± 0.31
HiRA	70.73 ± 0.18	85.33 ± 0.14	80.64 ± 0.33	74.39 ± 0.21	87.60 ± 0.13	83.20 ± 0.11	93.64 ± 0.02	84.16 ± 0.18	82.44 ± 0.10
MoSA	72.56 ± 0.17	85.63 ± 0.14	80.35 ± 0.32	**76.25** ± 0.47	88.14 ± 0.11	82.77 ± 0.19	94.10 ± 0.08	85.57 ± 0.08	83.17 ± 0.10
SHARE	**74.06** ± 0.35	**86.41** ± 0.33	**80.72** ± 0.16	76.08 ± 0.79	**89.33** ± 0.39	**85.85** ± 0.26	**94.44** ± 0.11	**85.69** ± 0.42	**84.07** ± 0.07
Llama-3-8B ($r = 32$)									
LoRA	72.05 ± 0.83	88.97 ± 0.24	81.73 ± 0.09	83.28 ± 0.65	92.83 ± 0.30	87.92 ± 0.35	96.22 ± 0.09	87.96 ± 0.33	86.37 ± 0.09
DoRA	72.15 ± 0.26	88.97 ± 0.39	81.71 ± 0.25	83.23 ± 0.31	92.95 ± 0.24	88.63 ± 0.80	96.12 ± 0.11	87.82 ± 0.09	86.45 ± 0.14
HiRA	75.14 ± 0.08	89.54 ± 0.08	82.06 ± 0.08	82.31 ± 0.13	92.52 ± 0.06	86.67 ± 0.42	96.12 ± 0.03	**89.45** ± 0.12	86.73 ± 0.06
MoSA	74.42 ± 0.99	**90.23** ± 0.92	82.10 ± 0.48	83.91 ± 1.04	93.30 ± 0.05	88.63 ± 0.52	96.30 ± 0.22	88.63 ± 0.43	87.19 ± 0.12
SHARE	**76.12** ± 0.03	90.18 ± 0.18	**82.17** ± 0.09	**85.35** ± 0.33	**93.72** ± 0.16	**89.77** ± 0.50	**96.62** ± 0.03	89.09 ± 0.05	**87.88** ± 0.09
Qwen3-4B ($r = 16$)									
LoRA	65.74 ± 0.32	86.60 ± 0.55	78.49 ± 0.26	88.24 ± 0.23	95.40 ± 0.27	88.12 ± 0.46	93.44 ± 0.09	82.89 ± 0.43	84.86 ± 0.19
DoRA	66.21 ± 0.43	86.87 ± 0.25	78.44 ± 0.47	88.35 ± 0.20	95.51 ± 0.16	88.32 ± 0.94	93.37 ± 0.09	82.95 ± 0.41	85.00 ± 0.17
HiRA	68.26 ± 0.66	87.63 ± 0.46	78.95 ± 0.36	88.33 ± 0.30	95.48 ± 0.13	88.89 ± 0.71	93.72 ± 0.07	82.64 ± 0.20	85.49 ± 0.12
MoSA	69.07 ± 0.92	87.86 ± 0.08	78.95 ± 0.29	**89.06** ± 0.34	95.42 ± 0.22	89.14 ± 0.39	93.88 ± 0.04	82.74 ± 0.09	85.76 ± 0.07
SHARE	**72.49** ± 0.19	**88.50** ± 0.24	**80.54** ± 0.37	88.64 ± 0.33	**95.83** ± 0.11	**89.49** ± 0.34	**94.39** ± 0.04	**85.80** ± 0.08	**86.96** ± 0.06

Table 2: GSM8K accuracy and HumanEval Pass@1 (%), reported as mean ± standard deviation over three seeds. Best and second-best means are shown in **bold** and underlined, respectively.

	Llama-2-7B ($r = 32$)		Llama-3-8B ($r = 32$)		Qwen3-4B ($r = 16$)	
Method	GSM8K	HumanEval	GSM8K	HumanEval	GSM8K	HumanEval
LoRA	44.51 ± 0.61	21.34 ± 1.61	71.18 ± 0.37	44.51 ± 0.61	82.07 ± 0.87	69.51 ± 0.61
DoRA	46.16 ± 1.47	21.54 ± 2.31	72.45 ± 0.43	45.12 ± 1.06	82.20 ± 0.46	70.53 ± 0.35
HiRA	55.56 ± 0.80	24.59 ± 0.35	75.80 ± 0.71	52.85 ± 1.96	83.91 ± 0.09	70.73 ± 1.83
MoSA	55.33 ± 0.69	25.61 ± 1.06	76.79 ± 0.72	55.49 ± 0.61	84.50 ± 0.51	71.14 ± 0.70
SHARE	**59.67** ± 0.43	**27.24** ± 0.93	**79.19** ± 0.42	**59.76** ± 0.62	**85.53** ± 0.27	**72.15** ± 0.35

4.2 Commonsense Reasoning Performance

Table 1 reports the results on the eight commonsense reasoning benchmarks. SHARE achieves the highest average accuracy on all three backbone models, reaching 84.07% on Llama-2-7B, 87.88% on Llama-3-8B, and 86.96% on Qwen3-4B. In each case, it outperforms all baselines. The improvements are also broadly distributed across individual datasets. SHARE achieves the best result on seven of the eight benchmarks with Llama-2-7B, six with Llama-3-8B, and seven with Qwen3-4B, ranking first in 20 of the 24 backbone–dataset combinations. Its average accuracy also shows low variation across random seeds, with standard deviations of 0.07, 0.09, and 0.06 on the three backbones, respectively. These results show that extensive parameter sharing across layers and modules can support strong high-rank adaptation performance. Despite using only a small number of shared parameter banks, SHARE consistently achieves higher average accuracy than methods that maintain separate trainable parameters for individual target weight matrices.

4.3 Mathematical Reasoning and Code Generation Performance

We further evaluate SHARE on mathematical reasoning and code generation. As shown in Table 2, SHARE achieves the best performance in all six backbone–task combinations. On GSM8K, it

Table 3: Training efficiency on a single NVIDIA A100 GPU. Throughput is measured in padded tokens per second. Best and second-best results are shown in **bold** and underlined, respectively.

	Llama-2-7B		Llama-3-8B		Qwen3-4B	
Method	Time (ms) ↓	Tokens/s ↑	Time (ms) ↓	Tokens/s ↑	Time (ms) ↓	Tokens/s ↑
LoRA	**782.8**	**3063**	**789.6**	**2616**	**611.4**	**3381**
DoRA	1344.6	1783	1462.4	1413	934.2	2213
HiRA	1046.8	2290	1082.8	1908	653.2	3165
MoSA	2798.2	857	2582.6	800	1497.0	1381
SHARE	1005.6	2384	1081.6	1910	739.6	2795

reaches 59.67%, 79.19%, and 85.53% with Llama-2-7B, Llama-3-8B, and Qwen3-4B, respectively. The improvement is particularly pronounced on Llama-2-7B, where SHARE improves over the strongest baseline from 55.56% to 59.67%. On HumanEval, SHARE achieves Pass@1 scores of 27.24%, 59.76%, and 72.15% on the three backbones. The largest gain is observed on Llama-3-8B, where it improves over MoSA from 55.49% to 59.76%. Together, these results show that SHARE performs consistently across backbones on both mathematical reasoning and code generation.

4.4 Training Efficiency

We compare the training efficiency of different adaptation methods on commonsense reasoning using a single NVIDIA A100 GPU with a batch size of 8. We report the average step time and training throughput over 50 steps. Since sequences are dynamically padded, throughput is calculated using the number of padded tokens actually processed in these steps. As shown in Table 3, LoRA achieves the highest throughput across all three backbone models. SHARE is substantially faster than MoSA, achieving $2.78\times$, $2.39\times$, and $2.02\times$ higher throughput on Llama-2-7B, Llama-3-8B, and Qwen3-4B, respectively. It also achieves higher throughput than DoRA on all three backbones and is comparable to HiRA on Llama-2-7B and Llama-3-8B. These gains come from constructing each shared update once and reusing it across multiple weight matrices. When the same update is applied to the same input, the corresponding matrix multiplication can also be reused. This reuse requires additional memory to store the fixed assignments and cached shared updates. The additional memory is approximately $1{,}062$ MiB for Llama-2-7B, $1{,}368$ MiB for Llama-3-8B, and 653 MiB for Qwen3-4B. Overall, SHARE reduces the training overhead of high-rank adaptation while introducing a moderate additional memory cost.

4.5 Rank Analysis of Learned Updates

We further analyze the learned updates on Llama-2-7B and compare SHARE with rank-32 LoRA. For SHARE, we examine the learned attention and FFN bank updates; for LoRA, we report the average stable rank across sampled layers and projections. As shown in Figure 2, LoRA updates have rank at

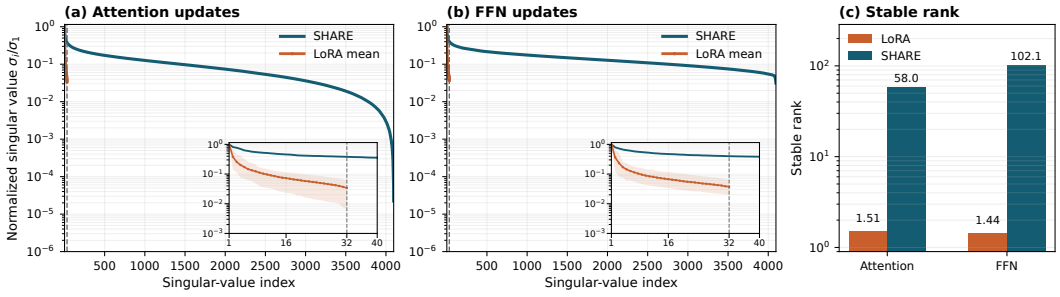

Figure 2: Rank analysis of learned updates on Llama-2-7B. Panels (a) and (b) compare the singular-value spectra of SHARE and rank-32 LoRA updates for attention and FFN, respectively. Panel (c) reports the stable rank of the learned updates.

most 32, whereas SHARE retains substantially broader singular-value spectra. To further quantify the effective dimensionality of the learned updates, we report the stable rank (Appendix A.2). The attention and FFN updates of SHARE achieve stable ranks of 58.0 and 102.1, respectively, compared with 1.51 and 1.44 for rank-32 LoRA. These results demonstrate that parameter sharing in SHARE does not collapse the learned updates into a low-dimensional subspace, allowing high-rank adaptation to be retained under extensive sharing.

4.6 Effect of Parameter Budget

We study how SHARE behaves under different parameter budgets. Figure 3 compares SHARE with LoRA using reference ranks from $r = 1$ to $r = 32$ on Llama-2-7B and Llama-3-8B. SHARE consistently outperforms LoRA under every matched budget, with particularly large gains at small values of r. At $r = 1$, SHARE already reaches average accuracies of 80.69% on Llama-2-7B and 86.34% on Llama-3-8B, close to the performance of LoRA at $r = 32$. Increasing the budget to $r = 2$ is sufficient for SHARE to surpass LoRA at $r = 32$ on both backbones. The advantage gradually narrows as the budget increases but remains consistent throughout the tested range. From $r = 1$ to $r = 32$, the average accuracy of SHARE increases by 3.38 points on Llama-2-7B and 1.61 points on Llama-3-8B, compared with gains of 8.73 and 4.66 points for LoRA. These results indicate that SHARE remains effective under very small parameter budgets and benefits less from increasing the budget than LoRA.

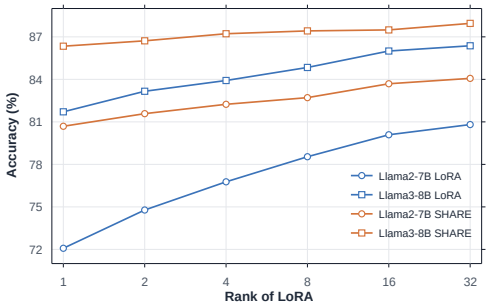

Figure 3: Effect of the parameter budget on commonsense reasoning. The value of r denotes the LoRA rank used to define the matched parameter budget.

4.7 Effect of Layer-wise Forward Drop

The shared parameterization of SHARE reuses the same updates across many Transformer layers, raising the question of whether every use must be evaluated at every training step. We therefore apply layer-wise forward drop: each layer independently skips its adaptation branch with probability p, while the pretrained computation remains unchanged. For retained layers, the adaptation output is scaled by $1/(1-p)$ to preserve its expected contribution, and all adaptation branches are enabled during evaluation. As shown in Figure 4, increasing p reduces the step time with a gradual loss in accuracy. At $p = 0.5$, the average accuracy decreases from 84.07% to 83.42%, while the step time decreases from 1005.6 ms to 868.2 ms. At $p = 0.75$, SHARE outperforms LoRA in both accuracy (81.80% vs. 80.73%) and step time (766.2 ms vs. 782.8 ms). These results show that forward drop provides a controllable accuracy–efficiency trade-off.

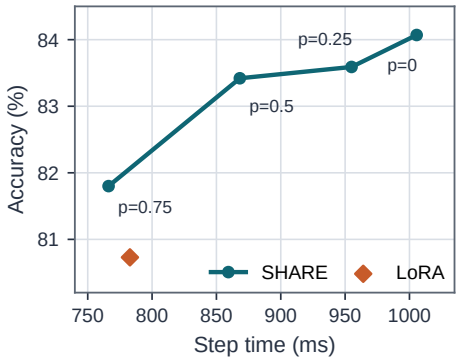

Figure 4: Accuracy–efficiency trade-off of layer-wise forward drop on Llama-2-7B.

5 Conclusion

In this paper, we introduced SHARE, a high-rank adaptation method that reuses a small number of trainable parameter banks across layers and modules. We provided a first-order analysis of the resulting weight updates and an efficient implementation that reduces redundant computation. Experiments across multiple backbones and downstream tasks show that SHARE consistently outperforms representative PEFT baselines while achieving competitive training efficiency. These results demonstrate that effective high-rank adaptation can be achieved with parameter sharing.

AI USE STATEMENT

Generative AI tools were used to support LaTeX structuring, language refinement, arithmetic checks, literature search, and table preparation. All AI-assisted text, calculations, and formatting outputs were subsequently reviewed by the authors. The authors take full responsibility for the accuracy of the final manuscript, including its claims, results, and presented artifacts.

REFERENCES

Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 1: long papers)*, pp. 7319–7328, 2021.

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 Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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.

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. *arXiv preprint arXiv:2107.03374*, 2021.

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, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Demi Guo, Alexander M Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. In *Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 1: Long papers)*, pp. 4884–4896, 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 *International conference on machine learning*, pp. 2790–2799. PMLR, 2019.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

Qiushi Huang, Tom Ko, Zhan Zhuang, Lilian Tang, and Yu Zhang. Hira: Parameter-efficient hadamard high-rank adaptation for large language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ting Jiang, Shaohan Huang, Shengyue Luo, Zihan Zhang, Haizhen Huang, Furu Wei, Weiwei Deng, Feng Sun, Qi Zhang, Deqing Wang, et al. Mora: High-rank updating for parameter-efficient fine-tuning. *arXiv preprint arXiv:2405.12130*, 2024.

Dawid Kopiczko, Tijmen Blankevoort, and Yuki Asano. Vera: Vector-based random matrix adaptation. In *International Conference on Learning Representations*, volume 2024, pp. 6815–6835, 2024.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pp. 3045–3059, 2021.

Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*, 2023.

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.

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.

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 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 4463–4473, 2019.

Yurun Song, Junchen Zhao, Ian G Harris, and Sangeetha Abdu Jyothi. Sharelora: Parameter efficient and robust large language model fine-tuning via shared low-rank adaptation. *arXiv preprint arXiv:2406.10785*, 2024.

Qwen Team et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 6(7):13, 2025.

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. *arXiv preprint arXiv:2307.09288*, 2023.

Xiequn Wang, Zhan Zhuang, Shengda Luo, and Yu Zhang. Mosa: Mosaic shared adaptation of large language models. In *The Fourteenth International Conference on Learning Representations*, 2026.

Zhengbo Wang, Jian Liang, Ran He, Zilei Wang, and Tieniu Tan. Lora-pro: Are low-rank adapters properly optimized? In *International Conference on Learning Representations*, volume 2025, pp. 93787–93808, 2025.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. In *International Conference on Learning Representations*, volume 2024, pp. 45040–45061, 2024.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

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. *arXiv preprint arXiv:2303.10512*, 2023.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

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 *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 12834–12859, 2024.

A Implementation Details

A.1 Parameter-Budget Allocation

The shared parameterization of SHARE substantially reduces the number of separate trainable parameter sets. In our main comparisons, however, we do not use this reduction to obtain a smaller parameter budget. Instead, we match the total number of trainable parameters to that of a rank-r LoRA configuration applied to the same target weight matrices.

For a weight matrix $W_s \in \mathbb{R}^{d_s^{\text{out}} \times d_s^{\text{in}}}$, rank-$r$ LoRA introduces $r(d_s^{\text{out}} + d_s^{\text{in}})$ trainable parameters. We therefore define the reference parameter budget as

$$K_{\text{ref}} = \sum_{s \in \mathcal{S}} r \left(d_s^{\text{out}} + d_s^{\text{in}} \right). \tag{12}$$

We distribute this budget across parameter banks according to the sizes of their shared updates. Let $M_b = m_b n_b$ denote the number of entries in the shared update D_b for bank b. Before rounding, the parameter budget allocated to bank b is

$$K_b^* = K_{\text{ref}} \frac{M_b}{\sum_{c=1}^{B} M_c}. \tag{13}$$

The final value K_b is obtained by rounding K_b^* to a hardware-friendly multiple and clipping it to the range $1 \le K_b \le M_b$. Consequently,

$$\sum_{b=1}^{B} K_b \approx K_{\text{ref}}. \tag{14}$$

Since K_b is approximately proportional to M_b, the average number of entries in D_b assigned to each trainable parameter,

$$\frac{M_b}{K_b}, \tag{15}$$

remains approximately consistent across banks. This allocation enables a parameter-matched comparison between SHARE and LoRA while maintaining a balanced parameter density across banks with different update sizes.

A.2 Stable Rank

Given a weight update ΔW with singular values $\{\sigma_i\}$, we measure its effective spectral dimension using the stable rank:

$$\text{srank}(\Delta W) = \frac{\|\Delta W\|_F^2}{\|\Delta W\|_2^2} = \frac{\sum_i \sigma_i^2}{\sigma_1^2}. \tag{16}$$

Unlike the matrix rank, which depends only on the number of nonzero singular values, the stable rank also reflects how spectral energy is distributed across singular directions. A larger stable rank therefore indicates that the update energy is distributed over a broader set of directions.

A.3 Construction of Fixed Coefficients

For each target weight matrix $s \in \mathcal{S}_b$, we assign a fixed coefficient α_s. In the default configuration, we first sample

$$\widetilde{\alpha}_s = \xi_s \exp(\epsilon_s), \qquad \xi_s \sim \text{Unif}\{-1, +1\}, \qquad \epsilon_s \sim \mathcal{U}[-\rho, \rho], \tag{17}$$

where ξ_s determines the sign of the coefficient, with positive and negative signs sampled with equal probability, and ρ controls the variation in coefficient magnitude.

We then normalize the coefficients by their root mean square:

$$\alpha_s = \frac{\widetilde{\alpha}_s}{\sqrt{\frac{1}{|\mathcal{S}_b|} \sum_{t \in \mathcal{S}_b} \widetilde{\alpha}_t^2}}. \tag{18}$$

After normalization, the coefficients within each bank satisfy

$$\frac{1}{|\mathcal{S}_b|}\sum_{s\in\mathcal{S}_b}\alpha_s^2 = 1. \tag{19}$$

The coefficients are sampled once before training and remain fixed throughout training.

B Understanding the Behavior of SHARE

B.1 Comparison with the LoRA Parameterization

Section 3.2 shows that SHARE defines a fixed linear parameterization $\Delta\mathbf{w} = P\boldsymbol{\theta}$, where P is determined by the parameter assignment and sharing pattern. Under gradient descent, the induced first-order change in the weight updates is

$$\delta\Delta\mathbf{w} = -\eta PP^\top\mathbf{g}, \tag{20}$$

where $\mathbf{g} = \nabla_{\Delta\mathbf{w}}\mathcal{L}$ denotes the gradient with respect to the selected weight updates. We compare this behavior with the factorized parameterization used in LoRA.

First-Order Behavior of Parameterized Updates. Consider a differentiable parameterization $\Delta\mathbf{w} = f(\boldsymbol{\phi})$, where $\boldsymbol{\phi}$ denotes the trainable parameter vector, and let

$$J = \frac{\partial\Delta\mathbf{w}}{\partial\boldsymbol{\phi}} \tag{21}$$

denote the corresponding Jacobian. By the chain rule, a gradient-descent step on $\boldsymbol{\phi}$ induces the following first-order change in weight space:

$$\delta\Delta\mathbf{w} = -\eta JJ^\top\mathbf{g}. \tag{22}$$

The column space of J determines the weight-update directions reachable by the parameterization. The orthogonal projection of $\mathbf{g}$ onto this space is

$$\Pi_J\mathbf{g} = J(J^\top J)^\dagger J^\top\mathbf{g}, \tag{23}$$

where † denotes the Moore–Penrose pseudoinverse. Let $J = U_r\Sigma_r V_r^\top$ denote the compact singular value decomposition of J. Then

$$JJ^\top\mathbf{g} = U_r\Sigma_r^2U_r^\top\mathbf{g}, \qquad \Pi_J\mathbf{g} = U_rU_r^\top\mathbf{g}. \tag{24}$$

Thus, both quantities lie in the same reachable subspace, while $JJ^\top\mathbf{g}$ additionally scales the components along different singular directions according to the squared singular values of J. When all nonzero singular values are equal, the induced direction is proportional to the orthogonal projection of the weight-space gradient.

SHARE and LoRA. For SHARE, the Jacobian is the fixed matrix P. Since different trainable parameters are assigned to disjoint weight entries, the columns of P are mutually orthogonal and $P^\top P$ is diagonal. When all projection axes have the same squared ℓ_2 norm γ, i.e., $P^\top P = \gamma I$, we have

$$PP^\top\mathbf{g} = \gamma\Pi_P\mathbf{g}. \tag{25}$$

Thus, under the equal-norm condition, the SHARE update is aligned with the negative projected weight-space gradient, up to a global scaling factor. When the column norms differ, the update remains in the same reachable subspace but receives axis-dependent scaling.

LoRA instead parameterizes the weight update as $\Delta W = BA$, where A and B are trainable low-rank factors. Its first-order change is

$$\delta\Delta W = (\delta B)A + B(\delta A). \tag{26}$$

Therefore, the corresponding Jacobian depends on the current values of A and B. Unlike the fixed mapping P in SHARE, the Jacobian of LoRA evolves during optimization, and its singular values can therefore change over training. This results in a time-varying, direction-dependent scaling in $JJ^\top\mathbf{g}$.

Table 4: Cosine similarity between the induced first-order direction $JJ^\top \mathbf{g}$ and the projected weight-space gradient $\Pi_J \mathbf{g}$ in a controlled residual-MLP setting.

Method	Step 0	Step 100	Step 1000
SHARE (all-one)	1.000	1.000	1.000
SHARE (continuous, $\rho = 0.1$)	0.999	0.999	0.999
LoRA	0.973	0.641	0.753

Controlled Comparison. We evaluate this difference in a controlled residual-MLP setting. The network contains eight residual layers with hidden dimension 64, and all layers are adapted. We fix the total number of trainable parameters to 4096. For SHARE, a single 64×64 update is parameterized and reused across all layers. For LoRA, each layer uses an independent rank-4 adapter under the same total parameter budget. The teacher network is constructed by applying independently sampled dense perturbations to the frozen base network.

At different training stages, we compute $\mathbf{g}$ with respect to the selected weight updates. We then measure the cosine similarity between $JJ^\top \mathbf{g}$ and its orthogonally projected counterpart $\Pi_J \mathbf{g}$:

$$\text{Cosine} = \frac{(JJ^\top \mathbf{g})^\top (\Pi_J \mathbf{g})}{\|JJ^\top \mathbf{g}\|_2 \|\Pi_J \mathbf{g}\|_2}. \tag{27}$$

As shown in Table 4, SHARE maintains near-perfect alignment with the projected weight-space gradient throughout training. The all-one coefficient setting gives exact alignment, as predicted by Eq. (25). The continuous coefficient variant achieves nearly identical alignment, indicating that this behavior is not restricted to the all-one coefficient setting.

In contrast, LoRA exhibits larger deviations from the projected direction as training progresses. This is consistent with its factorized parameterization, for which the Jacobian changes with the evolving factor values and can introduce non-uniform scaling across reachable directions.

Overall, this analysis highlights a geometric property of the SHARE parameterization beyond its ability to construct high-rank updates. Its fixed linear mapping keeps the correspondence between shared-parameter updates and reachable weight-space directions fixed throughout optimization, whereas the corresponding mapping in LoRA evolves with its trainable factors.

B.2 Sharing a Single Parameter Bank across Modules

We further examine a more aggressive sharing configuration in which all target weight matrices reuse a single trainable parameter vector, regardless of their update shapes. Unlike the default SHARE configuration, which uses separate parameter banks for different update shapes, this variant uses one parameter vector for the query, key, value, up, and down weight matrices.

Let

$$\boldsymbol{\theta} = [\theta_1, \ldots, \theta_K]^\top \tag{28}$$

denote the shared trainable parameter vector. Each target weight matrix s has its own fixed assignment $\kappa_s(i, j)$ that maps entries of its update matrix to elements of $\boldsymbol{\theta}$:

$$\Delta W_s[i, j] = \alpha_s \theta_{\kappa_s(i,j)}. \tag{29}$$

Because each target weight matrix uses its own assignment, matrices with different update shapes can reuse the same trainable parameter vector.

Table 5 compares this single-bank configuration with the default update-shape-based configuration on Llama-2-7B. Sharing one parameter vector across all target weight matrices retains similar downstream performance, with the average accuracy decreasing from 84.07 to 83.75. However, the step time increases from 1005.6 ms to 3539.4 ms, and throughput decreases from 2384 to 677 tokens/s.

The small difference in accuracy indicates that trainable parameters can be shared across attention and feed-forward weight matrices even when their update shapes differ. The substantial efficiency

Table 5: Comparison of the default update-shape-based configuration and single-bank sharing on Llama-2-7B. Average denotes the mean accuracy across the eight commonsense reasoning datasets. Throughput is measured in padded tokens per second. The best result is shown in **bold**.

Configuration	BoolQ	PIQA	SIQA	ARC-C	ARC-E	OBQA	HellaSwag	WinoG	Avg.	Time (ms)	Tokens/s
Default	**74.06**	**86.41**	**80.72**	**76.08**	**89.33**	85.85	**94.44**	**85.69**	**84.07**	**1005.6**	**2384**
Single Bank	73.82	86.16	79.78	75.58	88.50	**86.17**	94.34	85.65	83.75	3539.4	677

gap, however, motivates the update-shape-based grouping used in the default configuration. When weight matrices with the same update shape reuse the same shared update, the update can be cached and, when the inputs are also shared, the corresponding matrix multiplication can be reused. In the single-bank configuration, each target weight matrix uses a different assignment and therefore constructs a different update matrix from the same parameter vector, preventing this computational reuse.

B.3 Extending the Parameter Budget with Layer Segments

When a parameter bank is shared across all Transformer layers, its number of trainable parameters is bounded by the number of entries in its shared update. For a bank with update shape $m_b \times n_b$, this gives $K_b \leq m_b n_b$, which places an upper bound on the parameter budget when the same bank is shared across the full model depth.

To support larger parameter budgets, we divide the Transformer layers into contiguous segments and assign an independent copy of each parameter bank to every segment. The assignment of weight matrices to banks according to update shape remains unchanged within each segment; only the depth range over which each bank is shared is reduced. For example, on Llama-2-7B, two segments use separate sets of banks for layers 0–15 and 16–31, while four segments divide the 32 layers into four groups of eight layers.

Each segment uses the same bank configuration as the original model-wide sharing setting. Consequently, increasing the number of segments approximately scales the total parameter budget in proportion to the number of segments. Starting from a parameter budget matched to rank-32 LoRA, two and four segments correspond approximately to the budgets of rank-64 and rank-128 LoRA, respectively.

Table 6 reports a small-scale evaluation on GSM8K with Llama-2-7B. Increasing the number of segments improves GSM8K accuracy from 59.67 with one segment to 61.29 with two segments and 61.67 with four segments. These results show that larger parameter budgets can be supported by shortening the sharing range across Transformer depth while retaining the same parameterization within each segment.

Table 6: Layer-segment scaling on Llama-2-7B. The LoRA reference rank indicates the approximate matched parameter budget. The best result is shown in **bold**, and the second-best result is underlined.

LoRA Reference Rank	Segments	GSM8K
32	1	59.67
64	2	<u>61.29</u>
128	4	**61.67**

C Additional Ablations

C.1 Effect of Learning Rate

We evaluate SHARE with learning rates of 1×10^{-5}, 2×10^{-5}, and 4×10^{-5}. As shown in Table 7, SHARE performs similarly with learning rates of 1×10^{-5} and 2×10^{-5}, achieving average accuracies of 84.07% and 84.14%, respectively. Increasing the learning rate to 4×10^{-5} reduces the

average accuracy to 82.92%. Overall, SHARE remains stable over a moderate learning-rate range, while a larger learning rate leads to a noticeable performance decrease.

Table 7: Learning-rate ablation on Llama-2-7B commonsense reasoning (%). The best result is shown in **bold**, and the second-best result is underlined.

Learning Rate	BoolQ	PIQA	SIQA	ARC-C	ARC-E	OBQA	HellaSwag	WinoG	Avg.
1×10^{-5}	**74.06**	**86.41**	80.72	76.08	89.33	**85.85**	**94.44**	85.69	84.07
2×10^{-5}	74.03	86.22	**80.84**	**76.58**	**89.58**	85.80	94.37	**85.73**	**84.14**
4×10^{-5}	73.88	85.52	80.79	73.75	86.71	83.14	94.11	85.42	82.92

C.2 Effect of Projection-Axis Construction

Table 8: Ablation of fixed coefficients on Llama-2-7B commonsense reasoning (%). The best result is shown in **bold**, and the second-best result is underlined.

Coefficient Setting	BoolQ	PIQA	SIQA	ARC-C	ARC-E	OBQA	HellaSwag	WinoG	Avg.
All-one	73.73	86.22	**80.79**	76.50	**89.54**	85.61	94.39	85.26	84.01
Random sign	73.48	85.93	80.34	**76.62**	88.85	85.52	94.29	**86.56**	83.95
Continuous ($\rho = 0.1$)	**74.06**	**86.41**	80.72	76.08	89.33	85.85	**94.44**	85.69	**84.07**
Continuous ($\rho = 0.3$)	73.35	86.14	80.28	75.08	88.97	**86.33**	94.39	86.25	83.85
Continuous ($\rho = 0.5$)	**74.06**	86.16	80.49	76.00	88.25	85.42	**94.44**	85.73	83.82

As shown in Section 3.2, each trainable parameter is associated with a projection axis given by the corresponding column $\mathbf{p}_j$ of P. With the parameter assignment and sharing pattern fixed, the coefficients α_s determine the relative signs and scales with which a shared parameter contributes to different weight matrices, thereby changing the corresponding projection axes.

We therefore compare several choices of α_s, including all-one coefficients, random signs, and continuous coefficients generated with different values of ρ. We use ρ=0.1 by default. As reported in Table 8, all five settings achieve similar average accuracies, ranging from 83.82% to 84.07%. The continuous setting with $\rho = 0.1$ achieves the highest average accuracy of 84.07%, while the all-one and random-sign settings achieve 84.01% and 83.95%, respectively. Increasing ρ from 0.1 to 0.5 also produces only small changes in average performance. Overall, SHARE shows limited sensitivity to the tested projection-axis constructions.

C.3 Detailed Results of Layer-wise Forward Drop

Table 9 reports the per-dataset results for the layer-wise forward-drop experiment on Llama-2-7B. During training, each Transformer layer independently skips its adaptation branch with probability p, while the pretrained computation remains unchanged. Retained adaptation outputs are scaled by $1/(1-p)$, and all adaptation branches are enabled during evaluation. These results complement the accuracy–efficiency comparison reported in Figure 4.

Table 9: Detailed results of layer-wise forward drop on Llama-2-7B. Average denotes the mean accuracy across the eight commonsense reasoning datasets. The best result is shown in **bold**, and the second-best result is underlined.

Drop Prob. p	BoolQ	PIQA	SIQA	ARC-C	ARC-E	OBQA	HellaSwag	WinoG	Average
0	**74.06**	**86.41**	80.72	**76.08**	**89.33**	**85.85**	**94.44**	85.69	**84.07**
0.25	73.25	85.79	**80.84**	75.08	87.88	85.61	93.94	86.34	83.59
0.50	72.89	85.68	**80.84**	74.75	87.67	84.85	93.99	**86.73**	83.42
0.75	70.74	83.17	78.81	72.67	86.92	82.20	93.86	86.03	81.80

Moderate forward drop causes only small changes on most datasets. At $p = 0.5$, the average accuracy decreases from 84.07 to 83.42, while SIQA and WinoGrande remain comparable to, or slightly above, the no-drop setting. Increasing the drop probability to 0.75 produces a larger degradation across datasets, reducing the average accuracy to 81.80.

A Second-Order View of Forward Drop. We further examine how layer-wise forward drop changes the training objective. Let U_l denote the output of the adaptation branch at Transformer layer l, and let $q = 1 - p$ denote the keep probability. During training, we independently sample $m_l \sim \text{Bernoulli}(q)$ across layers and replace U_l with

$$\widetilde{U}_l = z_l U_l, \qquad z_l = \frac{m_l}{q}. \tag{30}$$

Since $\mathbb{E}[z_l] = 1$, the adaptation output is preserved in expectation:

$$\mathbb{E}[\widetilde{U}_l] = U_l. \tag{31}$$

Because the subsequent network computation is nonlinear, however, this does not imply that the resulting loss or gradient is an unbiased estimate of its full-forward counterpart.

Let $\mathcal{L}(z_1, \ldots, z_L; \boldsymbol{\theta})$ denote the training loss as a function of the layer-wise scaling variables, where $\boldsymbol{\theta}$ denotes the trainable parameters and the full-forward model corresponds to $z_l = 1$ for all l. Let $\mathcal{L}_{\text{full}} = \mathcal{L}(\mathbf{1}; \boldsymbol{\theta})$ and $\mathcal{L}_{\text{drop}} = \mathcal{L}(\mathbf{z}; \boldsymbol{\theta})$. Defining $\epsilon_l = z_l - 1$, we have

$$\mathbb{E}[\epsilon_l] = 0, \qquad \mathbb{E}[\epsilon_l^2] = \frac{p}{1-p}. \tag{32}$$

A second-order expansion around the full-forward point $\mathbf{z} = \mathbf{1}$ gives

$$\mathcal{L}(\mathbf{1} + \boldsymbol{\epsilon}; \boldsymbol{\theta}) \approx \mathcal{L}(\mathbf{1}; \boldsymbol{\theta}) + \sum_l \frac{\partial \mathcal{L}}{\partial z_l} \epsilon_l + \frac{1}{2} \sum_{l,k} \frac{\partial^2 \mathcal{L}}{\partial z_l \partial z_k} \epsilon_l \epsilon_k, \tag{33}$$

where the derivatives are evaluated at $\mathbf{z} = \mathbf{1}$.

Since the masks are sampled independently across layers, $\mathbb{E}[\epsilon_l] = 0$ and $\mathbb{E}[\epsilon_l \epsilon_k] = 0$ for $l \neq k$. Taking the expectation therefore gives

$$\mathbb{E}_m[\mathcal{L}_{\text{drop}}] \approx \mathcal{L}_{\text{full}} + \frac{p}{2(1-p)} \sum_{l=1}^{L} \left. \frac{\partial^2 \mathcal{L}}{\partial z_l^2} \right|_{\mathbf{z}=\mathbf{1}}. \tag{34}$$

Thus, although the adaptation output is preserved in expectation, forward drop changes the expected training objective through the curvature of the loss with respect to the layer-wise adaptation scale. When these curvature terms are positive, the additional term favors solutions whose loss is less sensitive to whether the adaptation branch is active at an individual layer.

The perturbation strength increases with the drop probability: $p/(1-p)$ equals $1/3$, 1, and 3 for $p = 0.25$, 0.5, and 0.75, respectively. This is consistent with the empirical trend that moderate forward drop preserves most of the downstream performance, whereas the more aggressive $p = 0.75$ setting produces a substantially larger degradation.

D LIMITATIONS

Our experiments focus on three backbone models and a limited set of downstream tasks, so the results may not fully reflect all model families or application settings. In addition, the efficiency of SHARE depends on the amount of update reuse available for a given architecture. Extending the method to broader model structures and training settings is left for future work.