RaSTA: Random Subspace Tuning Adaptation

Anonymous authors
Paper under double-blind review

Abstract

Parameter-efficient fine-tuning (PEFT) methods such as LoRA specialize a pretrained model by learning a small additive update to its frozen weights. However, LoRA's trainable parameter count is tied to the shape of each adapted matrix, costing at least the sum of its input and output dimensions even at rank one. We propose Random Subspace Tuning Adaptation (RaSTA), which adapts the model within frozen random subspaces shared across layers and learns only how their directions interact, allowing trainable parameter budgets well below rank-one LoRA without storing the frozen directions in the adapter checkpoint. Its two variants trade off the expressivity of the interactions against the number of directions reached under a matched budget. RaSTA-DM learns a dense mixing matrix between the bases, while RaSTA-HS fixes the mixing as a Walsh–Hadamard transform and learns one scaling vector per basis. To our knowledge, this is the first use of the Walsh–Hadamard transform for this purpose. The latter is somewhat more costly but performs better and degrades more gracefully as the budget shrinks. Across mathematical reasoning, commonsense reasoning, and code generation with 7–8B models, as well as natural-language understanding with RoBERTa-large, both variants remain competitive with rank-one LoRA, and sometimes with higher-rank LoRA, at budgets ranging from less than half to about one tenth of its parameter count. Both variants also train faster and use less peak memory than rank-one LoRA, except for RaSTA-HS at its largest considered budget.

1 Introduction

Fine-tuning is the standard way of specializing pretrained large language models and other foundation models for downstream tasks. Full fine-tuning is expensive in both compute and memory, especially when one base model is adapted separately for many tasks or users, because each task-specific checkpoint stores its own full set of model parameters. Parameter-efficient fine-tuning (PEFT) reduces these costs by updating only a small fraction of the parameters. Earlier PEFT methods often introduced inference latency or could not match the accuracy of full fine-tuning (Lialin et al., 2023; Han et al., 2024).

Low-rank adaptation (LoRA) (Hu et al., 2022), now the default PEFT method, addresses both problems. It adapts weight matrices in Transformer attention and feed-forward modules by expressing each update as a learned low-rank product. Specifically, LoRA freezes pretrained weight $W \in \mathbb{R}^{m\times n}$, and learns an additive update $\Delta W = BA$, where $B \in \mathbb{R}^{m\times r}$ and $A \in \mathbb{R}^{r\times n}$ are trainable, with $r \ll \min(m, n)$. This factorization is motivated by the low intrinsic dimensionality of fine-tuning updates (Aghajanyan et al., 2021). ΔW has rank at most r and can be merged into W at inference, thereby introducing no additional inference latency. Despite its simplicity and strong empirical results, LoRA's trainable parameter count remains tied to the dimensions of the adapted weight, since every matrix requires $r(m+n)$ parameters. Even at rank one, this amounts to $m+n$ parameters, and competitive performance often requires a larger rank. Thus, as model dimensions grow, LoRA requires increasingly large task-specific adapters, together with greater training-memory and optimization costs. These costs are compounded when one model is adapted to many tasks or users.

To reduce the parameter cost imposed by LoRA's dependence on model dimensions, subsequent work has explored more compact ways to represent the update ΔW. Some retain LoRA's factorization but change how its factors are parameterized, tying them across layers (Renduchintala et al., 2023) or constructing them from a small shared bank (Li et al., 2024). Others reshape the constraints on the update itself by learning scaling or mixing coefficients over frozen bases (Kopiczko et al., 2024;

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.

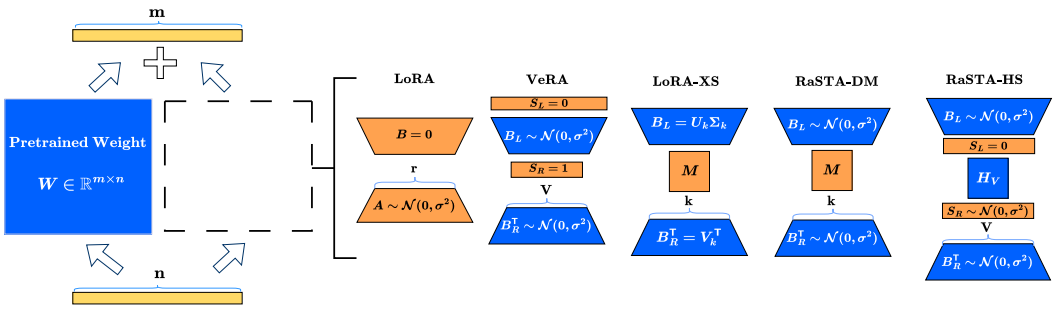


Figure 1: **RaSTA in relation to its predecessors.** Each right-hand side diagram shows how one method factorizes the additive update (dashed) to the frozen pretrained weight $W \in \mathbb{R}^{m \times n}$, with frozen components in blue, trainable components in orange, and initializations annotated. LoRA trains both low-rank factors of rank r; VeRA freezes a shared random basis pair and trains two scaling vectors; LoRA-XS freezes bases obtained from the top-k SVD of the weight and trains the dense core M; RaSTA-DM trains the dense core between shared random bases instead; RaSTA-HS fixes the core to the Walsh–Hadamard matrix H_V and trains only the two scaling vectors.

Bałazy et al., 2024; Abbasi Koohpayegani et al., 2024), or synthesizing the update from a learned sparse spectrum under a fixed transform (Gao et al., 2024).

Following the frozen-basis approach, we propose **RaSTA** (**ra**ndom **s**ubspace **t**uning **a**daptation), which reduces the dependence of the trainable parameter count on adapted layer dimensions by learning a small mixing core within fixed random bases. Particularly, RaSTA's formulation builds on VeRA (Kopiczko et al., 2024) and LoRA-XS (Bałazy et al., 2024). It combines LoRA-XS's mixing-core structure with VeRA's use of shared frozen random bases, as illustrated in Figure 1. We express RaSTA's update as

$$\Delta W = \gamma \, B_L \, \widetilde{M} \, B_R^\top, \tag{1}$$

where $B_L \in \mathbb{R}^{m \times k}$ and $B_R \in \mathbb{R}^{n \times k}$ are frozen random Gaussian bases shared across compatible layers which define the available left and right directions, $\widetilde{M}$ is the effective mixing core which controls the interactions between the basis directions from B_L and B_R, and γ scales the update.

We study two ways to allocate parameters within this formulation. **RaSTA-DM** (dense mixing) learns every entry of a dense core $\widetilde{M} = M \in \mathbb{R}^{k \times k}$, using k directions per basis and k^2 trainable parameters per adapted matrix. It shares LoRA-XS's dense-core structure and differs only in the frozen bases: RaSTA-DM uses random bases shared across compatible layers, whereas LoRA-XS derives them from a truncated SVD of each pretrained weight. Random shared bases avoid per-weight SVD at initialization, reduce the memory to store bases during training, and can be dropped from adapter checkpoints entirely and regenerated from stored seeds—simpler and more reproducible than recomputing SVDs to reconstruct LoRA-XS's bases. Our experiments show the two choices perform comparably on average, suggesting spectral bases are not necessary for this family.

In the second variant, **RaSTA-HS** (Hadamard scaling), we place a fixed, normalized Walsh–Hadamard matrix between two learned diagonal scalings: $\widetilde{M} = \mathrm{diag}(s_L) \, H_V \, \mathrm{diag}(s_R)$. Here H_V provides dense mixing between the left and right basis directions while $s_L, s_R \in \mathbb{R}^V$ scale their contributions, giving $2V$ trainable parameters for V directions per side. Whereas RaSTA-DM learns k^2 pairwise interactions among k directions, RaSTA-HS learns $2V$ scalings around a fixed mixing pattern, reaching more directions at the same budget. We use the Walsh–Hadamard matrix for its orthogonal rows and equal-magnitude entries, which couple every left–right direction pair with the same magnitude before the learned scalings apply. Prior methods apply fixed transforms over a weight's coordinates to synthesize updates from learned sparse spectra (Gao et al., 2024; Shen et al., 2025; Jeon et al., 2026); RaSTA-HS instead places the transform inside the mixing core, between the frozen shared bases—to our knowledge, the first use of the Walsh–Hadamard transform in this role.

We introduce two design choices for allocating parameters and scaling updates. *First*, a compression hyperparameter ρ sets the trainable parameter budget for each $m \times n$ weight matrix to approximately

$\sqrt{mn}/\rho$. [1] This gives larger weight matrices more parameters, as LoRA does implicitly. The basis widths therefore vary with the dimensions of the adapted weights, rather than remaining uniform across layers as in the original VeRA and LoRA-XS formulations. RaSTA-DM and RaSTA-HS choose their basis sizes so that $k^2 \approx 2V \approx \sqrt{mn}/\rho$. Under this matched budget, RaSTA-DM learns unrestricted mixing among fewer directions (k), whereas RaSTA-HS uses fixed mixing to reach more directions (V).

Second, we set the update scale to $\gamma = \sqrt{\rho}$ to compensate for changes in update magnitude as the budget varies. This helps keep optimization behavior comparable when using a single learning rate per variant across budgets. The motivation is analogous to rank-stabilized LoRA (Kalajdzievski, 2023), which adjusts the update scale as the rank changes. We examine both design choices through ablation studies. Section 3 develops them in detail.

We evaluate on mathematical reasoning, commonsense reasoning, code generation, and natural-language understanding, across 7–8B language models and RoBERTa-large. Across a budget range from under half of rank-one LoRA's parameters down to roughly a tenth, both RaSTA variants stay competitive with low-rank LoRA and degrade gracefully as the budget shrinks. All code, experiment scripts, and instructions for retrieving benchmark datasets are publicly available at https://anonymous.4open.science/r/rasta-798F/.

In summary, our contributions are as follows: **1)** We formulate parameter-efficient adaptation as learned mixing inside frozen random subspaces shared across layers. The bases need not be stored in the adapter checkpoint, and the trainable budget is decoupled from the adapted layer's dimensions. **2)** We instantiate two variants exposing an expressivity–coverage tradeoff: dense mixing of a few directions (RaSTA-DM), versus a fixed Walsh–Hadamard mixing that scales many more directions at equal budget (RaSTA-HS). To our knowledge this is the first use of the Walsh–Hadamard transform inside the mixing core of a frozen-basis adapter. **3)** We introduce a size-aware budget hyperparameter ρ and a matched update scaling that keeps a single learning rate effective across budgets; ablations and training-loss analysis validate both. **4)** Across four task families, 7–8B models and RoBERTa-large, and budgets from under half down to a tenth of rank-one LoRA, both variants remain competitive with LoRA and degrade gracefully. We further profile time and memory, showing both variants train within memory budgets where LoRA, VeRA, and full fine-tuning run out.

2 Related Work

From parameter-efficient tuning to fixed-subspace adaptation. Early PEFT approaches updated selected pretrained parameters (Ben Zaken et al., 2022), inserted lightweight adapter modules (Houlsby et al., 2019), or optimized continuous prompts and prefixes (Lester et al., 2021; Li & Liang, 2021), establishing that adaptation need not modify the whole model—though some added inference latency or did not consistently match full fine-tuning (Lialin et al., 2023; Han et al., 2024). Relatedly, Aghajanyan et al. (2021) show fine-tuning has low intrinsic dimensionality: optimizing as few as 200 parameters in a random subspace recovers 90% of full-fine-tuning performance.

LoRA (Hu et al., 2022) brought these ideas into individual weight matrices, representing each update as a learned low-rank product; its simplicity, effectiveness, and lack of inference latency have made it a dominant PEFT approach. Subsequent variants target training memory (Dettmers et al., 2023), performance (Liu et al., 2024), and convergence via improved initialization or asymmetric learning rates (Meng et al., 2024; Hayou et al., 2024), all retaining LoRA's learned low-rank factors so that each update's row and column spaces adapt to the task. A separate lineage instead fixes some or all of the adapted subspaces and learns only coefficients within them, further reducing trainable parameters: VeRA (Kopiczko et al., 2024) shares frozen Gaussian bases across layers and learns layer-specific diagonal scalings; LoRA-XS (Bałazy et al., 2024) learns a small core between per-layer bases from a truncated SVD of the pretrained weight; and VB-LoRA (Li et al., 2024) forms updates as sparse top-k mixtures from a global vector bank.

RaSTA is most closely related to VeRA and LoRA-XS (Figure 1): like VeRA it uses weight-independent frozen Gaussian bases shared across layers of matching shape, and like LoRA-XS it places a learned core between left and right bases. They differ in which bases are fixed and where

[1] Ignoring rounding, setting $\rho = 1$ gives a budget of $\sqrt{mn} \leq (m+n)/2$ trainable parameters per weight matrix, at most half of the $m + n$ parameters required by rank-one LoRA.

the learned parameters sit. LoRA-XS learns a dense core within layer-specific SVD bases; VeRA parameterizes $\Delta W = \mathrm{diag}(s_L) B_L \, \mathrm{diag}(s_R) B_R^\top$, learning two scaling vectors ($s_L \in \mathbb{R}^m$ over output coordinates, $s_R \in \mathbb{R}^V$ over latent directions). RaSTA instead controls the mixing between random left and right directions, through either a learned dense core (RaSTA-DM) or diagonal scalings around a fixed Hadamard matrix (RaSTA-HS).

Random representations and fixed subspaces. RaSTA also follows a longer line of work showing that fixed random representations can provide an effective space for learning. Echo state networks (Jaeger, 2001) and extreme learning machines (Huang et al., 2006) demonstrate that fixed, randomly initialized networks can provide useful feature representations, with only a linear readout trained on these features. Random Fourier features (Rahimi & Recht, 2007) similarly approximate shift-invariant kernels through fixed randomized features. Together, these methods demonstrate that learning can remain effective when optimization is restricted to coefficients over a sufficiently rich random representation.

The intrinsic-dimensionality formulation of Aghajanyan et al. (2021) applies the principle of learning within a fixed random subspace directly to fine-tuning. They apply the random projection using the Fastfood transform (Le et al., 2013) to avoid storing a dense projection matrix. NOLA (Abbasi Koohpayegani et al., 2024) further brings random bases to matrix-level adaptation, decoupling the trainable parameter count from the rank and enabling parameter budgets below LoRA's rank-one minimum. It parameterizes each low-rank factor as a learned mixture of multiple frozen random matrices, whereas RaSTA learns a mixing core between two frozen random vector bases.

Fixed orthogonal transforms. The normalized Walsh–Hadamard matrix H_V is orthogonal, has entries of equal magnitude $1/\sqrt{V}$, and can be applied in $O(V \log V)$ time. In PEFT, fixed transforms have been used to synthesize ΔW over the coordinates of the weight matrix from a sparse set of learned spectral coefficients, using the discrete Fourier (Gao et al., 2024), Hartley (Shen et al., 2025), and cosine (Du et al., 2025) transforms. QWHA (Jeon et al., 2026) adopts the Walsh–Hadamard transform in this role for quantized models. Outside PEFT, fixed transforms combined with learned diagonal matrices form efficient structured linear layers. Fastfood transform speeds up random Fourier features by replacing the dense Gaussian matrix with products of Hadamard and random diagonal matrices, adaptive Fastfood (Yang et al., 2015) learns these diagonals, and ACDC (Moczulski et al., 2016) interleaves learned diagonals with the discrete cosine transform. RaSTA-HS similarly places learned diagonal scalings around a fixed transform, but uses the result as a mixing core between frozen bases in an additive update rather than as a standalone linear layer.

3 Method Details

Table 1 summarizes the update formulations and trainable parameter counts of the methods discussed in Sections 1 and 2. For a frozen pretrained weight matrix $W \in \mathbb{R}^{m \times n}$, each method learns an additive update ΔW. LoRA trains both factors of a low-rank update, with rank at most r. VeRA instead freezes random bases shared across layers and learns scalings of the output coordinates and V latent directions. LoRA-XS and RaSTA-DM learn a dense $k \times k$ mixing core between frozen bases. LoRA-XS derives its frozen bases from a truncated SVD of each pretrained weight matrix, so if $W = U \Sigma V^\top$, then $B_L = U_k \Sigma_k$ and $B_R = V_k$. Here, Σ_k contains the k largest singular values, and U_k and V_k contain the corresponding left and right singular vectors. RaSTA-DM instead uses shared random bases as VeRA does. RaSTA-HS also uses shared random bases, with width V, but learns two scaling vectors for the left and right latent directions around a fixed Hadamard core.

Basis construction and sharing. We generate RaSTA's left and right bases B_L and B_R independently by sampling i.i.d. $\mathcal{N}(0, 1)$ entries and normalizing each column to unit norm. Generation is deterministic given a seed key encoding the basis side, the corresponding input or output dimension, and the basis width; frozen bases are shared across all layers requiring the same full basis shape. They therefore need not be stored in adapter checkpoints: each layer stores only its learned parameters—the core M (RaSTA-DM) or scalings s_L, s_R (RaSTA-HS)—plus the seed keys, and the bases are regenerated by the same procedure when reloading or merging into W.

Mixing cores. Each entry $\widetilde{M}_{ij}$ of the effective core weights the outer product of the i-th left and j-th right basis vectors. Whereas LoRA adapts its update's row and column spaces, RaSTA learns combinations of directions within fixed left and right spans. RaSTA-DM learns the k^2 entries of an

Table 1: Update formulations and trainable parameter counts per adapted weight matrix. Bold symbols in the update formulas denote trainable components. γ is a fixed update scale.

Method	Update ΔW	Trainable components	Basis construction	Parameter count
LoRA (Hu et al., 2022)	$\boldsymbol{BA}$	B, A	N/A (factors are learned)	$r(m+n)$
VeRA (Kopiczko et al., 2024)	$\mathrm{diag}(\boldsymbol{s_L})B_L\,\mathrm{diag}(\boldsymbol{s_R})B_R^\top$	s_L, s_R	Shared random bases	$m+V$
LoRA-XS (Bałazy et al., 2024)	$B_L\boldsymbol{M}B_R^\top$	M	Per-weight SVD	k^2
RaSTA-DM	$\gamma\, B_L\boldsymbol{M}B_R^\top$	M	Shared random bases	k^2
RaSTA-HS	$\gamma\, B_L\,\mathrm{diag}(\boldsymbol{s_L})H_V\,\mathrm{diag}(\boldsymbol{s_R})B_R^\top$	s_L, s_R	Shared random bases	$2V$

unconstrained dense core $\widetilde{M} = M \in \mathbb{R}^{k\times k}$, initialized to 0. RaSTA-HS instead uses the structured core $\widetilde{M} = \mathrm{diag}(s_L)H_V\,\mathrm{diag}(s_R)$, with $H_V \in \mathbb{R}^{V\times V}$ a fixed normalized Walsh–Hadamard matrix and only the scalings $s_L, s_R \in \mathbb{R}^V$ learned ($2V$ parameters). We set $s_L = 0$ and draw s_R's entries i.i.d. from $\mathcal{N}(0, \sigma^2)$, $\sigma = 0.01$; VeRA likewise initializes $s_L = 0$ but sets s_R to a nonzero constant.

Hadamard transforms. RaSTA-HS uses a Hadamard transform to provide dense mixing within its core without explicitly storing a dense mixing matrix. For a power-of-two order n, the normalized Walsh–Hadamard matrix H_n is defined by recursive Sylvester construction:

$$H_{2n} = \frac{1}{\sqrt{2}}\begin{pmatrix} H_n & H_n \\ H_n & -H_n \end{pmatrix} \qquad H_1 = [1]. \tag{2}$$

The entries in H_n have equal magnitude $\pm 1/\sqrt{n}$ and its rows are mutually orthogonal, so $H_n H_n^\top = I_n$. Multiplication by H_n is the Walsh–Hadamard transform (WHT). The recursive structure of H_n in Eq. 2 allows the transform to be evaluated by the FWHT algorithm (Beauchamp, 1975) in $\mathcal{O}(n \log n)$ time, rather than the $\mathcal{O}(n^2)$ required when multiplying by a dense matrix.

RaSTA uses the normalized Walsh–Hadamard matrix H_V. Hadamard is the natural choice for this fixed factor: every mixed direction therefore carries the same norm and none is a linear combination of the others, so the learned scalings act on a uniformly weighted, non-redundant set of directions and cannot collapse onto a small subset of the subspace.

Parameter budget. The two variants spend parameters differently: RaSTA-DM learns k^2 interactions among k directions, whereas RaSTA-HS learns $2V$ scalings around a fixed interaction and can therefore use a much larger width V. Since the widths are not tied to the layer's dimensions, we set them explicitly through a single compression hyperparameter ρ, which fixes each layer's target budget to $\sqrt{mn}/\rho$—allocating more parameters to larger layers, in proportion to the geometric mean of their dimensions. The widths are

$$k = \left\lfloor \left(\frac{\sqrt{mn}}{\rho}\right)^{1/2} \right\rfloor, \qquad V = \min\Big(2^{\,\mathrm{round}\left(\log_2 \frac{\sqrt{mn}}{2\rho}\right)},\; 2^{\lfloor \log_2 \min(m,n)\rfloor}\Big), \tag{3}$$

so that $k^2 \approx 2V \approx \sqrt{mn}/\rho$. RaSTA-HS requires V to be a power of two for the transform H_V, hence the rounding, and caps it at the largest power of two not exceeding $\min(m, n)$ so the basis directions cannot outnumber the layer's rank. For a 4096×4096 layer at $\rho = 2$, RaSTA-DM uses $k = 45$ (2025 parameters) and RaSTA-HS uses $V = 1024$ (2048): nearly the same budget buys either unrestricted mixing among 45 directions or structured mixing across 1024.

Update scaling. Increasing ρ reduces the parameter budget and can also reduce the magnitude of the unscaled update. We set $\gamma = \sqrt{\rho}$ for both RaSTA variants to compensate for this effect. This scaling is intended to make comparisons across budgets at a common learning rate more meaningful, analogous in spirit to rank-stabilized LoRA (Kalajdzievski, 2023).

4 Experiments

Section 4.1 reports fine-tuning quality, first on three instruction-tuning tasks (mathematical reasoning, commonsense reasoning, and code generation) and then on natural-language understanding. Section 4.2 analyzes the results beyond the per-task tables: how quality degrades as the budget shrinks across the frozen-basis methods, and how the update scaling γ shapes the training dynamics. Section 4.3 profiles single-step training time and peak memory. Detailed hyperparameters, the full efficiency grid, and the fused-transform analysis are reported in the appendix.

Table 2: Trainable parameter counts for all methods and budgets on the instruction-tuning backbones. LoRA-XS has exactly the same count as RaSTA-DM at every ρ and is therefore omitted.

	LoRA (r)			RaSTA-HS (ρ)			RaSTA-DM (ρ)			VeRA
Model	**64**	**8**	**1**	**1**	**2**	**4**	**1**	**2**	**4**	V**=1024**
Llama-2-7B	159.9M	20.0M	2.50M	1.31M	655K	328K	1.15M	571K	285K	1.59M
Mistral-7B	167.8M	21.0M	2.62M	1.18M	590K	295K	1.12M	552K	274K	1.61M
Llama-3-8B	167.8M	21.0M	2.62M	1.18M	590K	295K	1.12M	552K	274K	1.61M

All experiments compare the two RaSTA variants at $\rho \in \{1, 2, 4\}$ against three baselines: LoRA uses ranks $r \in \{1, 8, 64\}$ for instruction tuning and $r \in \{1, 2, 8\}$ for GLUE (Wang et al., 2018), with scaling hyperparameter $\alpha = 2r$. VeRA uses basis widths $V = 1024$ and $V = 256$, respectively. LoRA-XS uses the same per-layer parameter allocation and update scale as RaSTA-DM, so the two differ only in their frozen bases: layer-specific SVD bases versus shared random bases.

All methods use AdamW, with learning rates fixed across pretrained models and budgets for each task. We use 2×10^{-4} for LoRA and 2×10^{-2} for VeRA throughout. On mathematical reasoning and code generation, RaSTA-HS uses 3×10^{-2}, while RaSTA-DM and LoRA-XS use 4×10^{-3}. On the remaining tasks, RaSTA-HS uses 2×10^{-2}, while RaSTA-DM and LoRA-XS use 2×10^{-3}. RaSTA and LoRA-XS use the scaling $\gamma = \sqrt{\rho}$ (Section 3). Each configuration is a single run at one fixed seed; in preliminary multi-seed runs we observed run-to-run spreads of one to two GSM8K (Cobbe et al., 2021) points, so differences of that magnitude should be read as within run-to-run variation. All experiments were run on a single H100-80GB GPU.

Table 3: Mathematical reasoning: GSM8K and MATH accuracy.

	Mistral-7B		Llama-2-7B		Llama-3-8B	
Method	**GSM8K**	**MATH**	**GSM8K**	**MATH**	**GSM8K**	**MATH**
LoRA r=64	74.45	21.00	56.03	9.00	77.10	25.20
LoRA r=8	71.65	20.40	47.50	6.10	76.35	24.60
LoRA r=1	67.85	16.40	41.35	5.20	71.27	21.60
RaSTA-HS ρ=1	66.26	18.00	43.20	5.40	71.11	20.80
RaSTA-HS ρ=2	65.31	15.40	42.46	4.60	69.98	20.40
RaSTA-HS ρ=4	63.91	14.80	38.59	4.40	68.46	17.20
RaSTA-DM ρ=1	66.26	16.40	38.13	5.20	69.60	19.00
RaSTA-DM ρ=2	61.71	16.20	34.19	3.40	65.58	20.80
RaSTA-DM ρ=4	59.89	14.20	30.71	3.60	64.75	19.40
LoRA-XS ρ=1	67.25	15.40	44.20	4.80	68.76	21.80
LoRA-XS ρ=2	63.46	16.40	39.42	3.80	66.94	17.80
LoRA-XS ρ=4	61.64	16.60	37.15	3.40	66.57	17.60
VeRA V=1024	68.31	18.40	46.10	5.20	73.31	21.60

4.1 Fine-Tuning Benchmarks

4.1.1 Instruction Tuning

We use three base models: Llama-2-7B (Touvron et al., 2023), Mistral-7B (Jiang et al., 2023), and Llama-3-8B (Dubey et al., 2024). All three are evaluated on mathematical reasoning and code generation, while commonsense reasoning is evaluated on Llama-3-8B only. Adapters are applied to all linear projections of the attention and MLP blocks; Table 2 reports the resulting trainable-parameter counts. Each model is trained for one epoch with a batch size of 128, using a cosine schedule with warmup ratio 0.03. Prompts follow the standard Alpaca format, and evaluation uses greedy decoding.

Mathematical reasoning. Models are fine-tuned on 100k samples of MetaMathQA (Yu et al., 2023) and evaluated on GSM8K and MATH (Hendrycks et al., 2021) with exact-match accuracy (Table 3).

Commonsense reasoning. Llama-3-8B is fine-tuned on Commonsense-170k (Hu et al., 2023) and evaluated on eight datasets: BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2021), ARC-e and ARC-c (Clark et al., 2018), and OBQA (Mihaylov et al., 2018) (Table 4).

Code generation. Models are fine-tuned on CodeFeedback (Zheng et al., 2024) and evaluated on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), using pass@1 accuracy (Table 5).

Table 4: Commonsense reasoning: per-dataset and average accuracy on Llama-3-8B.

Method	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg.
LoRA r=64	77.22	91.19	82.75	96.01	89.74	93.90	84.64	90.60	88.26
LoRA r=8	74.71	90.64	82.60	95.66	89.58	93.56	83.36	90.00	87.51
LoRA r=1	73.49	89.77	80.76	94.09	88.40	92.80	80.03	88.20	85.94
RaSTA-HS ρ=1	73.30	89.28	82.50	94.66	88.24	92.34	81.91	88.40	86.33
RaSTA-HS ρ=2	73.00	88.90	82.29	94.23	87.92	92.30	81.57	86.40	85.83
RaSTA-HS ρ=4	72.14	89.12	81.68	92.86	86.19	91.92	79.44	86.00	84.92
RaSTA-DM ρ=1	71.77	88.41	79.58	91.58	85.08	91.79	79.44	85.60	84.16
RaSTA-DM ρ=2	71.68	87.21	78.97	90.12	83.50	90.49	78.24	83.20	82.93
RaSTA-DM ρ=4	70.76	85.91	77.12	88.36	82.64	90.49	78.24	82.20	81.97
LoRA-XS ρ=1	71.99	88.14	82.19	93.59	87.29	92.05	81.31	86.20	85.34
LoRA-XS ρ=2	72.51	87.32	80.66	92.92	85.64	91.46	79.61	83.00	84.14
LoRA-XS ρ=4	70.64	86.62	80.50	91.96	85.48	90.99	79.95	82.40	83.57
VeRA V=1024	72.94	89.01	81.99	94.28	87.53	92.21	80.63	87.40	85.75

Table 6: Results of fine-tuning RoBERTa-large on tasks in GLUE benchmark. Numbers are Matthews correlation for CoLA, Spearman correlation for STS-B, and accuracy for the remaining tasks; Avg. is the mean across the eight tasks. Params is the trainable-parameter count (classification head excluded). LoRA-XS matches RaSTA-DM at every ρ.

Method	Params	CoLA	SST-2	MRPC	STS-B	QQP	MNLI	QNLI	RTE	Avg.
LoRA r=8	786K	67.32	95.76	89.46	91.85	91.50	90.53	94.71	83.75	88.11
LoRA r=2	197K	67.57	95.53	89.46	90.94	90.96	90.29	94.67	82.67	87.76
LoRA r=1	98K	66.27	95.87	89.22	90.52	90.25	90.00	94.20	84.12	87.56
RaSTA-HS ρ=1	49.2K	66.09	96.44	89.46	91.17	89.72	89.38	94.27	80.51	87.13
RaSTA-HS ρ=2	24.6K	62.77	95.87	89.22	90.44	89.53	89.33	93.78	82.31	86.65
RaSTA-HS ρ=4	12.3K	59.67	96.22	88.73	90.38	89.45	88.37	93.32	80.51	85.83
RaSTA-DM ρ=1	49.2K	57.53	95.53	89.46	89.91	88.51	88.21	92.48	77.62	84.91
RaSTA-DM ρ=2	23.2K	56.58	94.72	86.52	88.03	88.27	87.42	91.23	75.45	83.53
RaSTA-DM ρ=4	12.3K	54.77	94.04	86.27	86.95	87.90	86.40	90.59	70.40	82.17
LoRA-XS ρ=1	49.2K	63.56	95.18	87.99	90.15	89.57	89.24	93.89	80.51	86.26
LoRA-XS ρ=2	23.2K	61.57	95.87	85.54	88.89	89.31	88.70	93.15	80.14	85.40
LoRA-XS ρ=4	12.3K	58.80	95.18	85.54	88.61	89.36	88.12	92.40	76.90	84.36
VeRA V=256	61.4K	59.71	95.41	87.64	91.47	89.56	89.62	94.75	78.34	85.81

4.1.2 Natural Language Understanding

We fine-tune RoBERTa-large (Liu et al., 2019) on the eight tasks of the GLUE benchmark. Adapters are applied to query and value projections only. The classification head is trainable in every run at a learning rate of 5×10^{-4}. Training uses AdamW linear schedule and 0.06 warmup ratio (Table 6).

4.2 Analysis and Discussion

Comparing frozen-basis methods across budgets. The per-task tables report many single-seed numbers, making overall trends hard to read. For a more robust view of how the three frozen-basis methods (RaSTA-HS, RaSTA-DM, LoRA-XS) compare as the budget shrinks, we normalize within each (model, task) cell—dividing every (method, ρ) score by the best any of the three attains at any budget on that cell—and average the fractions across models. Each curve thus reads as the fraction of the best achievable frozen-basis performance a method retains at a given budget.

Table 5: Code generation: HumanEval and MBPP pass@1 accuracy.

	Mistral-7B		Llama-2-7B		Llama-3-8B	
Method	HumanEval	MBPP	HumanEval	MBPP	HumanEval	MBPP
LoRA r=64	53.66	61.90	29.27	38.36	55.49	67.20
LoRA r=8	50.00	58.99	22.64	36.57	53.66	67.20
LoRA r=1	44.51	58.73	21.34	35.45	46.34	67.72
RaSTA-HS ρ=1	44.51	57.14	20.73	35.71	47.56	66.67
RaSTA-HS ρ=2	42.68	57.41	20.73	35.45	47.56	63.49
RaSTA-HS ρ=4	40.85	56.35	16.46	34.39	48.78	66.67
RaSTA-DM ρ=1	43.29	56.61	16.46	32.80	46.95	62.96
RaSTA-DM ρ=2	41.46	56.35	16.46	35.71	46.34	64.81
RaSTA-DM ρ=4	40.24	54.50	15.24	32.01	49.39	59.79
LoRA-XS ρ=1	39.02	57.67	15.85	29.37	48.17	60.32
LoRA-XS ρ=2	37.80	59.79	15.85	28.84	42.68	60.32
LoRA-XS ρ=4	38.41	56.35	16.46	30.42	45.73	61.11
VeRA V=1024	42.07	53.70	19.51	35.45	49.39	63.76

Figure 2 shows the result, with fixed-budget LoRA r=1 and VeRA overlaid under the same normalization. Two points stand out. First, RaSTA-HS dominates at every budget, and its ρ=2 configuration matches or exceeds RaSTA-DM and LoRA-XS at ρ=1—similar or better accuracy at roughly half the parameters, and, as the profiling shows, no added runtime or memory cost. Second, RaSTA-HS degrades most gracefully, its curve the flattest of the three as ρ grows, while the dense mixer is consistently the most budget-sensitive.

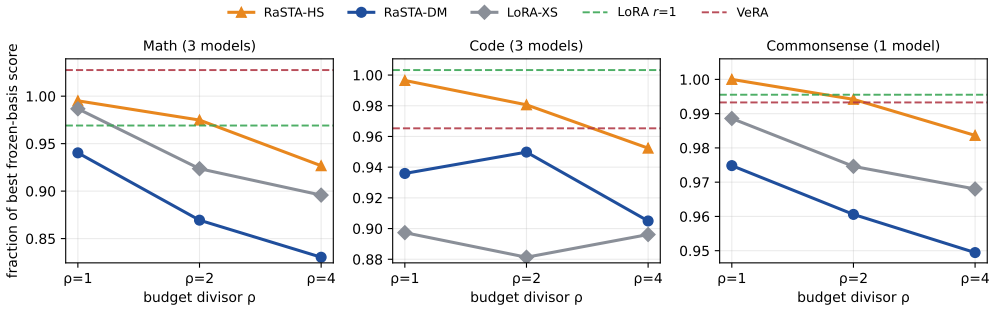


Figure 2: Frozen-basis methods across budgets. Scores are normalized by the best frozen-basis result within each (model, task) cell and averaged across models; dashed lines are the fixed-budget LoRA r=1 and VeRA baselines under the same normalization.

Training dynamics and the effect of update scaling. Figure 3 shows training-loss curves on Llama-2-7B for both RaSTA variants across budgets, with and without the budget-stabilized update scaling $\gamma(\rho) = \sqrt{\rho}$ (at ρ=1 the two coincide by construction). The scaling behaves as designed. Without scaling, the effective per-step update shrinks and the unscaled curves (dashed) descend consistently more slowly than their scaled counterparts, with the gap growing from ρ=2 to ρ=4. With scaling (solid), the curves at different budgets become near-parallel. Comparing variants, RaSTA-DM is more sensitive than RaSTA-HS both to the budget itself and to the absence of scaling, mirroring its position as the most budget-sensitive method in Figure 2.

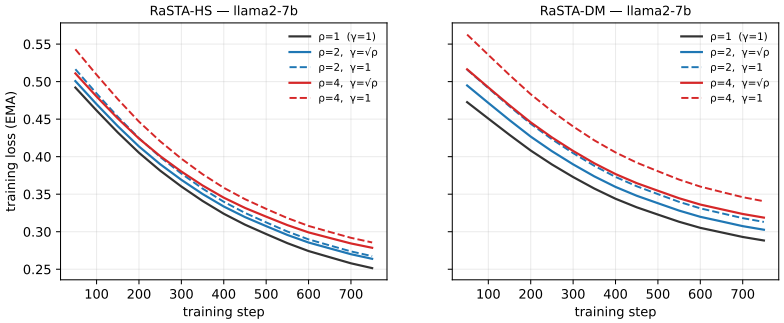
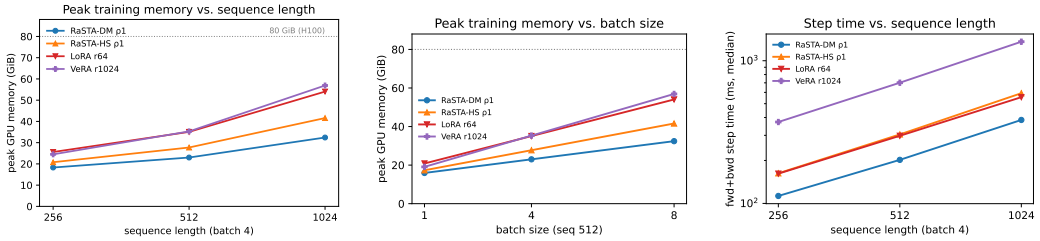


Figure 3: Training loss (Exponential Moving Average (EMA)) on Llama-2-7B for RaSTA-HS (left) and RaSTA-DM (right) at $\rho \in \{1, 2, 4\}$, with the update scaling γ=$\sqrt{\rho}$ (solid) and without (dashed); ρ=1 is identical in both cases. All runs share one learning rate per variant.

4.3 Single-Step Time and Memory Profile

We isolate the wall-clock and memory cost of a single optimization step, measured outside any trainer framework so that only the adapter and the core forward/backward/update path contribute. Each method is driven with random synthetic batches of fixed shape, and every statistic is the median over 30 CUDA-event-timed steps, each preceded by 10 warm-up steps to absorb one-time costs (kernel autotuning, allocator warm-up). Table 7 reports peak memory and step time on Mistral-7B; Figure 4 shows how both scale with sequence length and batch size, and Appendix B.1 gives the full grid across all three backbones.



(a) Peak memory vs. sequence length (batch 4). (b) Peak memory vs. batch size (seq 512). (c) Step time vs. sequence length (batch 4).

Figure 4: Scaling of peak memory and forward+backward step time on Mistral-7B for four representative configurations. Memory grows smoothly with sequence length (a) and batch size (b), preserving the method ordering with RaSTA-DM lowest throughout; step time (c) is near-linear in sequence length (log–log axes), with VeRA the consistent outlier. The gap widens with workload—at seq 1024 and batch 8, RaSTA-DM sits 20–25 GiB below the LoRA/VeRA pair—so the memory advantage is largest in the long-context, large-batch regime where the 80 GiB ceiling (dotted) binds.

Three patterns hold. *First*, on peak memory both RaSTA variants and LoRA-XS sit well below every LoRA and VeRA configuration; RaSTA-DM and LoRA-XS are nearly indistinguishable ($\approx$23 GiB, flat in ρ), and RaSTA-HS a little higher but still under the smallest LoRA rank. *Second*, raising ρ from 1 to 4 cuts RaSTA-HS's step time by roughly a third and lowers its memory, whereas RaSTA-DM and LoRA-XS stay flat—their per-step cost dominated by the frozen base rather than the small trainable factors. *Third*, and most consequential for long-context training, at seq 2048 RaSTA-DM, LoRA-XS, and the larger-ρ RaSTA-HS configurations still fit on the 80 GiB device, while full fine-tuning, every LoRA rank, and VeRA run out of memory. VeRA is the latency outlier throughout, several times slower than the RaSTA variants at matched settings. This ordering holds across all three backbones and the full sequence-length and batch-size sweep (Figure 4). RaSTA-HS applies its Walsh–Hadamard transform through a fused CUDA kernel, whose effect on step time we quantify in Appendix B.2.

Table 7: Training-step efficiency on Mistral-7B (batch 4, bf16 base, fp32 trainable parameters, AdamW): peak memory at seq 512 and 2048, and median forward+backward step time at seq 512 (30 CUDA-event-timed steps, 10 warm-up each; IQR $<$1%). "—" marks out-of-memory on the 80 GiB device.

Method	Pk_{512} (GiB)	t_{512} (ms)	Pk_{2048} (GiB)
Full fine-tuning	67.6	208	—
LoRA (r=64)	35.1	298	—
LoRA (r=8)	32.7	274	—
LoRA (r=1)	32.5	251	—
RaSTA-HS (ρ=1)	27.7	304	69.3
RaSTA-HS (ρ=2)	25.3	241	60.2
RaSTA-HS (ρ=4)	24.1	211	55.6
RaSTA-DM (ρ=1)	23.0	202	51.3
RaSTA-DM (ρ=2)	23.0	197	51.2
RaSTA-DM (ρ=4)	23.0	196	51.1
LoRA-XS (ρ=1)	23.4	198	51.6
LoRA-XS (ρ=2)	23.3	197	51.5
LoRA-XS (ρ=4)	23.2	192	51.3
VeRA (V=1024)	35.3	701	—

5 Conclusion

RaSTA treats a fine-tuning update as an interaction between frozen random bases shared across layers. Learning that interaction as a dense core already keeps pace with spectrally constructed bases; more surprisingly, it can be fixed to a Hadamard transform with only two learned scaling vectors—our strongest variant. Both fall well below LoRA's trainable count while training faster and in less memory.

Much of the method's potential remains to be explored. It keeps VeRA's outer structure, so refinements built on VeRA transfer directly; the bases could be shared more aggressively, and the budget knob ρ allocated unevenly across layers. It should also pair well with quantized backbones, and its seed-regenerable checkpoints suit serving one model to many tasks or users.

AI USE STATEMENT

The method proposed in this paper, the experimental design, and the claims we make were conceived and decided by the authors. We used a generative AI assistant in a supporting role, and we take responsibility for the final content of this work.

Among tasks for which ICLR requires disclosure, we used AI assistance for feedback on experimental and ablation design, for implementing parts of the experimental and analysis code, for interpreting results, and for checking the elementary mathematical statements in the method section. Among tasks for which disclosure is recommended, we also used it to help write and edit code, produce the figures, draft and polish portions of the text, and format references. We did not use AI to generate data, and tasks such as translation and qualitative data analysis do not apply to this work.

We reviewed all AI-assisted work: code was tested and verified against the raw experimental outputs, and all reported numbers, text, and interpretations were checked by the authors.

REFERENCES

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

Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Association for Computational Linguistics*, 2021.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.01772*, 2021.

Klaudia Bałazy, Mohammadreza Banaei, Rémi Lebret, Jacek Tabor, and Karl Aberer. LoRA-XS: Low-rank adaptation with extremely small number of parameters. *arXiv preprint arXiv:2405.17604*, 2024.

Kenneth G. Beauchamp. *Walsh Functions and Their Applications*. Academic Press, 1975.

Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Association for Computational Linguistics*, 2022.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, 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 (NAACL)*, 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, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, 2023.

Zhekai Du, Yinjie Min, Jingjing Li, Ke Lu, Changliang Zou, Liuhua Peng, Tingjin Chu, and Mingming Gong. LoCA: Location-aware cosine adaptation for parameter-efficient fine-tuning. In *International Conference on Learning Representations*, 2025.

Abhimanyu Dubey et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Ziqi Gao, Qichao Wang, Aochuan Chen, Zijing Liu, Bingzhe Wu, Liang Chen, and Jia Li. Parameter-efficient fine-tuning with discrete Fourier transform. In *International Conference on Machine Learning*, 2024. TODO verify eprint/author list.

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

Soufiane Hayou, Nikhil Ghosh, and Bin Yu. LoRA+: Efficient low rank adaptation of large models. In *International Conference on Machine Learning*, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *NeurIPS Datasets and Benchmarks*, 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*, 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 *International Conference on Learning Representations*, 2022.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. LLM-Adapters: An adapter family for parameter-efficient fine-tuning of large language models. In *Empirical Methods in Natural Language Processing*, 2023.

Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks. Technical Report 148, German National Research Center for Information Technology, 2001.

Hyesung Jeon, Seojune Lee, Beomseok Kang, Yulhwa Kim, and Jae-Joon Kim. QWHA: Quantization-aware Walsh-Hadamard adaptation for parameter-efficient fine-tuning on large language models. In *International Conference on Learning Representations*, 2026.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Lavril, et al. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

Damjan Kalajdzievski. A rank stabilization scaling factor for fine-tuning with LoRA. *arXiv preprint arXiv:2312.03732*, 2023.

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

Quoc Le, Tamás Sarlós, and Alexander Smola. Fastfood: Approximating kernel expansions in loglinear time. In *International Conference on Machine Learning*, 2013.

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

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Association for Computational Linguistics*, 2021.

Yang Li, Shaobo Han, and Shihao Ji. VB-LoRA: Extreme parameter efficient fine-tuning with vector banks. In *Advances in Neural Information Processing Systems*, 2024.

Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 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 *International Conference on Machine Learning*, 2024.

Yinhan Liu et al. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Fanxu Meng, Zhaohui Wang, and Muhan Zhang. PiSSA: Principal singular values and singular vectors adaptation of large language models. In *Advances in Neural Information Processing Systems*, 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 (EMNLP)*, 2018.

Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. ACDC: A structured efficient linear layer. In *International Conference on Learning Representations*, 2016.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, 2007.

Adithya Renduchintala, Tugrul Konuk, and Oleksii Kuchaiev. Tied-LoRA: Enhancing parameter efficiency of LoRA with weight tying. *arXiv preprint arXiv:2311.09578*, 2023. TODO verify eprint/venue.

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 (EMNLP)*, 2019.

Yixian Shen et al. SSH: Sparse spectrum adaptation via discrete Hartley transformation. *arXiv preprint*, 2025. TODO verify full author list, title, and arXiv id.

Hugo Touvron, Louis Martin, Kevin Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP*, 2018.

Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1476–1483, 2015.

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

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 (ACL)*, 2019.

Tianyu Zheng, Ge Zhang, Travis 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 Findings)*, 2024.

A Ablations

All ablations use mathematical reasoning (GSM8K, MATH) under the main learning-rate regime; the main-configuration numbers are the corresponding runs from Table 3.

A.1 Parameter Allocation

Our allocation sizes each layer's core from its own dimensions (Section 3), whereas prior frozen-basis methods use a single dimensionality for every layer. We compare against this uniform alternative at matched total budget (ρ=1): a single shared V/k chosen so that the total adapter count equals that of the per-layer rule, with γ fixed so that allocation is the only difference (Table 8). Per-layer allocation performs at least as well, improving on uniform allocation by 1.3 GSM8K and 1.4 MATH points on average. Its main advantage is structural: it makes a single budget knob ρ well-defined across layers of different shapes and removes the shared-size hyperparameter that the uniform scheme must re-tune for each architecture.

Table 8: Per-layer vs. uniform allocation at matched total budget (ρ=1, Mistral-7B).

	ρ-based allocation		**Uniform rank**	
Method	**GSM8K**	**MATH**	**GSM8K**	**MATH**
RaSTA-HS	66.26	18.00	65.96	15.00
RaSTA-DM	66.26	15.40	63.99	15.60
LoRA-XS	67.25	15.40	66.34	14.80

A.2 One-Sided Scaling

RaSTA-HS learns gain vectors on both sides of the Hadamard core. We compare against a one-sided variant that freezes $s_L \equiv 1$ and therefore has half as many trainable parameters. In the two-sided form, the gradient with respect to each gain vector is scaled by the other. Fixing $s_L \equiv 1$ removes this scaling, so the gradients with respect to s_R are larger and the main learning rate overshoots. We therefore evaluate the one-sided variant at smaller rates. Because it matches the parameter count of the two-sided model at ρ=2, we also report that configuration (Table 9).

At equal budget (ρ=1), the one-sided variant trails the two-sided one on both models and benchmarks, even at its best rate. The gap is small on Mistral-7B (66.41 vs. 68.08 GSM8K) and large on LLaMA-2-7B (35.56 vs. 44.50). At matched parameter count, spending the budget on both sides at a lower ρ is preferable to freezing one side.

Table 9: One-sided vs. two-sided RaSTA-HS. Params are relative to the two-sided model at ρ=1.

				Mistral-7B		**LLaMA-2-7B**	
Gains	**ρ**	**Params**	**lr**	**GSM8K**	**MATH**	**GSM8K**	**MATH**
Two-sided (main config)	1	$1\times$	main	66.26	18.00	43.20	5.40
Two-sided	2	$0.5\times$	main	65.31	15.40	42.46	4.60
One-sided	1	$0.5\times$	2×10^{-3}	65.13	16.60	30.33	3.20
	1	$0.5\times$	5×10^{-3}	66.41	15.40	35.56	4.00

B Efficiency Profiling Details

B.1 Full Efficiency Grid

Table 10 extends Table 7 to all three backbones: peak memory and combined forward+backward step time at batch 4 and seq 512, plus peak memory at seq 2048. The three patterns noted in the body hold on every model. The ρ knob steadily lowers both time and peak memory for RaSTA-HS, while RaSTA-DM and LoRA-XS are essentially flat in ρ. The memory ordering is stable across models, with both RaSTA variants and LoRA-XS below every LoRA and VeRA configuration at seq 512, and the gap widens at seq 2048, where only RaSTA-DM, LoRA-XS, and the larger-ρ RaSTA-HS configurations remain within the 80 GiB budget while full fine-tuning, every LoRA rank, and VeRA run out of memory. VeRA remains the latency outlier throughout, several times slower than the RaSTA variants at matched settings.

Table 10: Full efficiency grid: peak memory (GiB) and median forward+backward step time (ms) at batch 4, seq 512, and peak at seq 2048 ("—" = out of memory on the 80 GiB device). Fused WHT for RaSTA-HS.

	Mistral-7B			Llama2-7B			Llama3-8B		
	Pk_{512}	t_{512}	Pk_{2048}	Pk_{512}	t_{512}	Pk_{2048}	Pk_{512}	t_{512}	Pk_{2048}
Full FT	67.6	208	—	62.9	204	—	75.6	220	—
LoRA r=64	35.1	298	—	32.7	294	—	39.1	306	—
LoRA r=8	32.7	274	—	30.5	269	—	36.8	282	—
LoRA r=1	32.5	251	—	30.3	249	—	36.5	259	—
RaSTA-HS ρ=1	27.7	304	69.3	26.9	299	68.4	31.8	313	—
RaSTA-HS ρ=2	25.3	241	60.2	24.2	237	58.3	29.4	249	71.9
RaSTA-HS ρ=4	24.1	211	55.6	22.9	211	53.3	28.2	219	67.3
RaSTA-DM ρ=1	23.0	202	51.3	21.7	202	48.5	27.0	211	63.0
RaSTA-DM ρ=2	23.0	197	51.2	21.7	198	48.4	27.0	205	62.9
RaSTA-DM ρ=4	23.0	196	51.1	21.6	182	48.3	27.0	204	62.9
LoRA-XS ρ=1	23.4	198	51.6	22.0	193	48.8	27.4	207	63.4
LoRA-XS ρ=2	23.3	197	51.5	21.9	200	48.6	27.3	205	63.2
LoRA-XS ρ=4	23.2	192	51.3	21.8	181	48.5	27.2	200	63.1
VeRA r=1024	35.3	701	—	33.8	691	—	41.2	1126	—

B.2 Fused vs. Reference WHT

RaSTA-HS applies a Walsh–Hadamard transform (WHT) inside every adapted projection. Our pure-PyTorch reference uses the standard fast butterfly: for a transform of size V, it runs $\log_2 V$ rounds, each replacing every pair of elements by their sum and difference via reshape/slice/add-sub/concatenate. Every round launches several kernels that read and write the full activation tensor, and autograd records them all, so backward replays the chain in reverse. The fused implementation instead calls a dedicated off-the-shelf CUDA kernel (`fast_hadamard_transform`): one launch that reads the tensor once, computes the whole butterfly in on-chip memory, and writes once—and since the normalized Hadamard matrix is symmetric orthogonal ($H = H^\top = H^{-1}$), the same kernel serves backward. As the WHT is bandwidth-bound, these advantages compound into $\mathcal{O}(\log_2 V)$ fewer memory passes and no retained autograd tensors.

Table 11 reports the effect on *whole-model* step time; the transform-local speedup is necessarily larger. The gain is largest for the biggest transforms and, there, flat in sequence length: at ρ=0.5 the fused kernel is $2.10\times$ faster at both seq 512 and seq 2048. For smaller transforms the fused win shrinks as the sequence lengthens and bandwidth begins to dominate—at ρ=4 it falls from $1.83\times$ to $1.29\times$. All comparisons elsewhere in the paper use the fused kernel, RaSTA-HS's default; at first use the implementation self-checks it against the butterfly reference and falls back if the kernel is unavailable or numerically inconsistent.

Table 11: Fused Hadamard kernel vs. the pure-PyTorch (Sylvester butterfly) reference: median forward+backward step time (ms) of the *full* model and the resulting whole-step speedup (torch/fused). RaSTA-HS on Mistral-7B, batch 4; larger ρ corresponds to a smaller Hadamard transform.

	seq 512			seq 2048		
Budget	torch	fused	speedup	torch	fused	speedup
ρ=0.5	680	323	2.10×	2615	1243	2.10×
ρ=1	598	304	1.96×	2271	1176	1.93×
ρ=2	413	241	1.71×	1426	914	1.56×
ρ=4	387	211	1.83×	1018	791	1.29×

C Distribution of learned coefficients .

Figure 5 examines how evenly the learned magnitude is spread across each method's trainable values, plotting the cumulative share of ℓ_2 energy against the fraction of values, sorted by magnitude (an

i.i.d. Gaussian vector and a perfectly even spread are shown as references). Two points emerge. First, the frozen-basis methods — RaSTA-HS, RaSTA-DM, LoRA-XS, and VeRA — track the Gaussian reference almost exactly, on all backbones: no small subset of coefficients dominates, so the adapters make full use of their budget and offer no obvious post-hoc pruning shortcut. This is consistent with the bases being random and roughly isotropic, leaving no coordinate privileged. Second, LoRA deviates mildly toward concentration — its curves sit above the Gaussian reference, more so at $r{=}8$ than $r{=}1$ — plausibly because both factors are learned, letting optimization allocate energy unevenly across directions of its own choosing. The shape is invariant to ρ (we show $\rho{=}1$), indicating that the budget knob changes capacity, not the character of the learned solution.

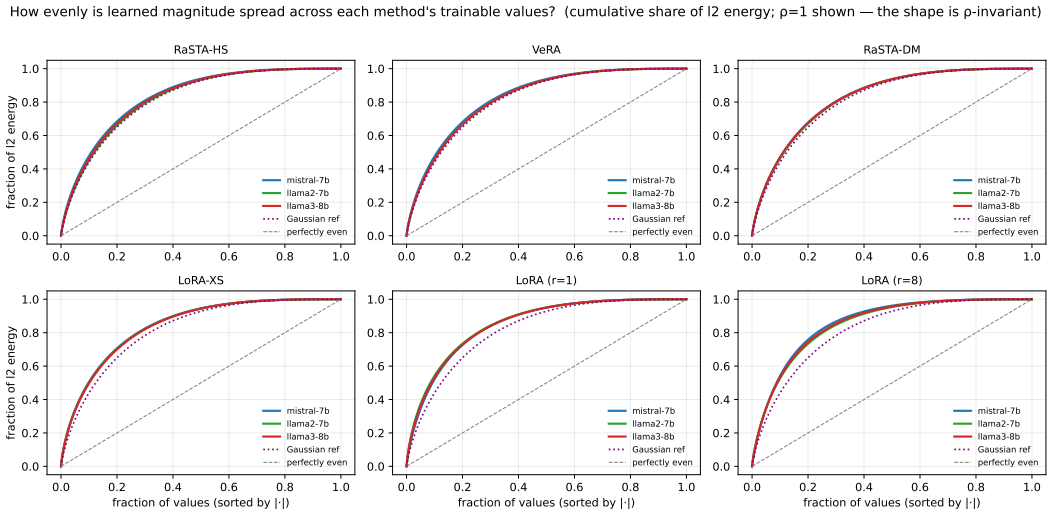


Figure 5: Cumulative share of ℓ_2 energy across trainable values, sorted by magnitude ($\rho{=}1$ shown; the shape is ρ-invariant). All methods remain close to the Gaussian reference and far from the diagonal of a sparse solution: learned adaptations are dense in their available directions.

D Training and Evaluation Details

Beyond the setup of Section 4: all runs use weight decay 0.01 and default AdamW moments, with gradient accumulation to the stated effective batch size. On the generative tasks, base weights are held in bf16 while trainable parameters and optimizer states are fp32; GLUE trains fully in fp32. Maximum sequence lengths are 512 (math, code), 256 (commonsense), and 128 (GLUE). On GLUE, MRPC, CoLA, RTE, and QQP train for 20 epochs and the remaining tasks for 10, and the reported number is the best dev-set metric across epochs.

Generation at evaluation is capped at 512 new tokens for GSM8K, 2048 for MATH, and 1024 for the code benchmarks, which are scored with the EvalPlus harness. The commonsense evaluation follows the protocol of Hu et al. (2023): beam search with 4 beams, temperature 0.1, 32 new tokens, and answers extracted by pattern matching.