LOOPEDLORA: LOW-RANK ADAPTATION WITH RECURRENT SUBSPACE DEPTH

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-Rank Adaptation (LoRA) is a widely used parameter-efficient fine-tuning method. A common way to improve LoRA adaptation is to increase the rank, which also increases the number of trainable parameters; despite many variants, most adapters retain a single-pass mapping through the low-rank bottleneck. We propose **LoopedLoRA**, which introduces weight-shared recurrent depth within this subspace. LoopedLoRA recurrently processes an r-dimensional latent state using a fusion-friendly RMSNorm–SiLU operator, adding only 0.22% more trainable parameters than LoRA at $r = 16$ on Qwen3-4B-Base. To allow the contribution of recurrent processing to vary across tokens, a learned latent blender adaptively combines the direct and recurrent paths, and LoopedLoRA-Light reuses its score to bypass recurrence at inference. Across four Qwen3 and Llama-3 models, LoopedLoRA improves GSM8K by up to 12.96 percentage points over LoRA and outperforms rank-doubled LoRA ($r = 32$) and competitive baselines on mathematical reasoning, while remaining competitive on code generation and commonsense reasoning. With fused Triton kernels and CUDA graph replay, full LoopedLoRA maintains decode latency comparable to LoRA, while LoopedLoRA-Light achieves up to 6.2% faster decoding.

1 INTRODUCTION

Low-Rank Adaptation (LoRA; Hu et al., 2022) is a widely used approach for parameter-efficient fine-tuning of large language models. It adapts a frozen projection through two low-rank matrices with few trainable parameters and low storage cost. However, standard LoRA remains a single-pass linear transformation: each token passes through an r-dimensional bottleneck only once.

This design raises a simple question:

"Should LoRA adaptation be improved only by widening the r-dimensional bottleneck?"

Although increasing the rank r is a common way to enhance LoRA adaptation, it linearly increases the number of trainable parameters. Recent variants improve adaptation through directional updates (Liu et al., 2024), improved initialization (Meng et al., 2024), weight sharing (Kang & Yin, 2026), feature stabilization (Wu et al., 2026), and nonlinear transformations (Dong et al., 2025; Luo et al., 2026). However, most retain a single-pass mapping through the low-rank bottleneck. This motivates another approach to improving adaptation: increasing *computational depth* within the low-rank space. The benefit of recurrent refinement may also differ across tokens, suggesting that its contribution should be weighted token-wise.

Recurrent architectures increase computational depth through parameter reuse. Early work introduced recurrence across Transformer depth (Dehghani et al., 2019; Giannou et al., 2023), while recent language models repeatedly apply Transformer stacks (Zhu et al., 2025; Lab et al., 2026) or organize backbone layers into recurrent middle-cycle blocks (Bae et al., 2025; Zeitoun et al., 2026). These approaches introduce recurrence into backbone hidden representations and modify the backbone architecture. We instead confine recurrent depth to the low-rank adapter bottleneck, enabling iterative state refinement while leaving the pre-trained backbone unchanged.

We propose **LoopedLoRA**, which introduces weight-shared recurrent computation into low-rank adaptation. LoopedLoRA encodes an input representation into an r-dimensional latent state, refines

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.

it for k iterations using a fusion-friendly RMSNorm–SiLU operator, and decodes it back to the output space. Increasing k adds no recurrent parameters; at $r = 16$, LoopedLoRA adds only 0.22% more trainable parameters than standard LoRA on Qwen3-4B-Base. A learned **latent blender** combines the direct and recurrent paths for each token, and **LoopedLoRA-Light** reuses its blending score to bypass recurrence below a threshold at inference time without additional training.

We evaluate LoopedLoRA on Qwen3-4B-Base, Qwen3-8B-Base, Llama-3.2-3B, and Llama-3.1-8B across mathematical reasoning, code generation, and commonsense reasoning benchmarks. LoopedLoRA achieves the largest gains on mathematical reasoning, improving GSM8K by up to 12.96 percentage points and MATH-500 by up to 3.20 percentage points over standard LoRA. It also improves code-generation performance by up to 2.22 percentage points, while remaining competitive on code generation and commonsense reasoning. These gains are obtained with only 0.22% more trainable parameters than LoRA at $r = 16$.

Our main contributions are summarized as follows:

- **Recurrent subspace depth.** We propose **LoopedLoRA**, which introduces weight-shared recurrence within the low-rank bottleneck, increasing computational depth with negligible parameter overhead ($+0.22\%$ at $r = 16$ on Qwen3-4B-Base).
- **Adaptive blending and efficient inference.** We introduce a latent blender that adaptively combines the direct and recurrent paths, and **LoopedLoRA-Light**, which selectively bypasses recurrence at inference time without additional training.
- **Empirical and efficiency evaluation.** Across four Qwen3 and Llama-3 models, LoopedLoRA achieves the largest gains on mathematical reasoning while remaining competitive on code generation and commonsense reasoning. A fused Triton kernel mitigates the decode-time overhead of recurrent computation.

2 Related Work

2.1 Low-Rank Adaptation and Its Variants

Low-Rank Adaptation (LoRA; Hu et al., 2022) parameterizes weight updates through a low-rank decomposition, $\Delta W = \frac{\alpha}{r} BA$. Subsequent variants improve LoRA adaptation through different mechanisms. DoRA (Liu et al., 2024) decouples magnitude and directional updates, PiSSA (Meng et al., 2024) initializes adapters with principal singular components, MiSS (Kang & Yin, 2026) uses shard sharing for efficient adaptation, and Stable-LoRA (Wu et al., 2026) stabilizes early feature learning through dynamic weight shrinkage.

Recent work further introduces nonlinear transformations into the low-rank path. AuroRA (Dong et al., 2025) inserts an adaptive nonlinear layer between the two projectors, while RanLoRA (Luo et al., 2026) combines nonlinear activation with residual-aware modulation. However, both remain single-pass feed-forward mappings. In contrast, LoopedLoRA repeatedly refines the low-dimensional representation using a weight-shared recurrent operator.

2.2 Looped and Recurrent Architectures

Looped architectures increase computational depth through parameter reuse. Early work introduced recurrent processing or cross-layer parameter sharing in Transformers (Dehghani et al., 2019; Lan et al., 2020), while later studies showed that looped Transformers can execute or learn iterative algorithms (Giannou et al., 2023; Yang et al., 2024). More recent language models reuse Transformer stacks across recurrent depth (Zhu et al., 2025; Lab et al., 2026) or apply recurrence to middle backbone blocks (Bae et al., 2025; Zeitoun et al., 2026).

These approaches introduce recurrence directly into backbone hidden representations and modify the backbone architecture itself. LoopedLoRA instead confines recurrence to the r-dimensional low-rank adapter subspace, leaving the pre-trained backbone unchanged. This enables iterative latent refinement within a parameter-efficient fine-tuning framework.

3 BACKGROUND

3.1 LOW-RANK ADAPTATION (LORA)

Low-Rank Adaptation (LoRA; Hu et al., 2022) parameterizes the update of a frozen linear projection using a low-rank decomposition. For a weight matrix $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, the forward pass is:

$$h = W_0 x + \Delta W x = W_0 x + \frac{\alpha}{r} BAx, \tag{1}$$

where $x \in \mathbb{R}^{d_{\text{in}}}$ is the input representation, $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $B \in \mathbb{R}^{d_{\text{out}} \times r}$ are the down- and up-projection matrices, respectively, and $r \ll \min(d_{\text{in}}, d_{\text{out}})$. The factor $\frac{\alpha}{r}$ controls the scale of the LoRA update.

Typically, A is randomly initialized while B is initialized to zero, yielding $\Delta W = 0$ at the start of training. Although LoRA substantially reduces the number of trainable parameters, its adapter branch remains a single-pass linear mapping. Increasing the rank r increases its parameter count linearly but does not introduce additional computational depth.

3.2 LOOPED ARCHITECTURES

Looped architectures increase computational depth by repeatedly applying a shared transformation (Dehghani et al., 2019; Giannou et al., 2023). Rather than stacking k distinct layers with independent parameters, a shared operator $\mathcal{F}_\theta$ is applied for k recurrent steps:

$$h^{(t)} = \mathcal{F}_\theta \left(h^{(t-1)} \right), \quad t \in \{1, \ldots, k\}, \tag{2}$$

where $h^{(0)}$ is the initial representation and k denotes the recurrent depth.

Because θ is shared across iterations, the parameter count is independent of k, allowing computational depth to increase without introducing separate parameters for each step. The intermediate representation is iteratively refined through these recurrent updates. Existing looped Transformers typically apply recurrence to backbone hidden representations; in Section 4, we instead confine recurrent computation to the low-rank adapter subspace.

4 METHOD

In this section, we present **LoopedLoRA**, which introduces recurrent subspace depth into low-rank adaptation. Rather than increasing the adapter rank, LoopedLoRA repeatedly refines a latent representation within the low-rank subspace using shared parameters. Figure 1 provides an overview of standard LoRA, LoopedLoRA, and LoopedLoRA-Light.

4.1 LOOPEDLORA: ARCHITECTURE AND FORMULATION

For a frozen weight matrix $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, standard LoRA applies the single-pass update $\Delta W x = \frac{\alpha}{r} BAx$. LoopedLoRA extends this formulation with three components operating in an r-dimensional low-rank subspace:

- **Subspace Encoder** $A \in \mathbb{R}^{r \times d_{\text{in}}}$: maps the input representation x to the initial latent state $z_0 = Ax \in \mathbb{R}^r$.
- **Shared Recurrent Processor** $R \in \mathbb{R}^{r \times r}$: iteratively refines the latent state for k steps using shared parameters.
- **Subspace Decoder** $B \in \mathbb{R}^{d_{\text{out}} \times r}$: maps the resulting latent state z_{out} back to the output space.

The resulting forward pass is

$$h = W_0 x + \frac{\alpha}{r} B z_{\text{out}}. \tag{3}$$

We use $r = 16$, $k = 4$, and $\alpha = 16$ as the default configuration and study the effect of k in Section 5.4.

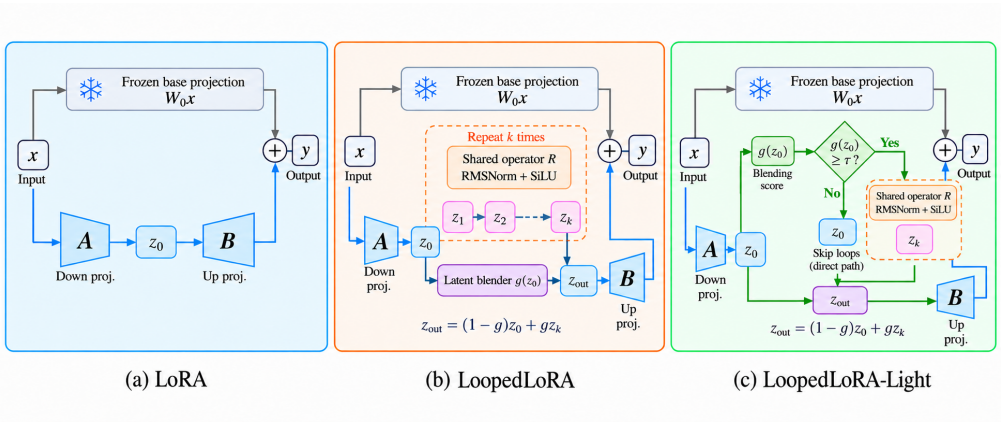


Figure 1: **Overview of LoopedLoRA. (a)** Standard LoRA maps the input through a single low-rank path. **(b)** LoopedLoRA repeatedly refines the low-dimensional latent state using a shared operator and blends it with the direct path. **(c)** LoopedLoRA-Light uses the learned blending score to bypass recurrent refinement below a threshold τ at inference time.

All recurrent computation is confined to the low-rank adapter subspace, leaving the pre-trained backbone unchanged. When $k = 0$ or the recurrent path is bypassed, $z_{\text{out}} = z_0$, and the formulation reduces to standard LoRA:

$$h = W_0 x + \frac{\alpha}{r} BAx. \tag{4}$$

4.2 Recurrent Subspace Refinement

Starting from $z_0 = Ax$, LoopedLoRA applies the shared recurrent operator for k steps:

$$z_t = \text{SiLU}\Big(R \cdot \text{RMSNorm}(z_{t-1})\Big), \quad t = 1, \ldots, k, \tag{5}$$

where $R \in \mathbb{R}^{r \times r}$ is shared across all iterations and $\text{SiLU}(u) = u \cdot \sigma(u)$ introduces nonlinearity. Before each recurrent transformation (RMSNorm; Zhang & Sennrich, 2019), we apply RMS normalization:

$$\text{RMSNorm}(v) = \frac{v}{\sqrt{\frac{1}{r}\sum_{i=1}^{r} v_i^2 + \epsilon}} \odot \gamma, \tag{6}$$

where $\gamma \in \mathbb{R}^r$ is a learnable gain initialized to $\mathbf{1}$. We initialize the recurrent matrix R with Kaiming initialization (He et al., 2015). RMS normalization controls the input scale at each recurrent step and improves the stability of repeated nonlinear transformations. Because the same R is reused across all steps, increasing k increases computational depth without introducing additional recurrent parameters.

Fusion-Friendly Operator Design. The recurrent operator is designed to admit efficient kernel fusion. For the rank-16 configuration, each Triton program processes one token's latent state and executes the fixed-depth recurrent loop within a single kernel invocation. Each step consists only of RMS normalization, an $r \times r$ matrix–vector product, and element-wise SiLU, while preserving the latent dimension. The intermediate latent states therefore need not be materialized in global memory between recurrent steps, avoiding repeated kernel launches and intermediate global-memory reads and writes. This is particularly useful for small bottleneck dimensions such as $r = 16$, where kernel-launch and memory-access overhead can dominate the small arithmetic cost. We evaluate the fused Triton implementation (Tillet et al., 2019) in Section 5.5.

4.3 Latent Blending

Different tokens may benefit differently from recurrent refinement. We therefore introduce a lightweight **latent blender** $g(z_0) \in (0, 1)$ conditioned on the initial latent state:

$$g(z_0) = \sigma\left(w^\top z_0 + b\right), \tag{7}$$

where $w \in \mathbb{R}^r$ and $b \in \mathbb{R}$ are learnable parameters. Because the blender operates directly in the low-rank space, it introduces only $r + 1$ parameters per adapted projection.

The blender continuously interpolates between the direct LoRA path z_0 and the recurrently refined state z_k:

$$z_{\text{out}} = (1 - g(z_0))z_0 + g(z_0)z_k = z_0 + g(z_0)(z_k - z_0). \tag{8}$$

Thus, the contribution of the recurrent path can vary across tokens while retaining the standard LoRA path. We evaluate the effectiveness of this learned blending in Section 5.4.

LoRA-Biased Initialization. We initialize $w = \mathbf{0}$ and $b = -2.0$, giving $g_{\text{init}} = \sigma(-2.0) \approx 0.119$. LoopedLoRA therefore starts with a stronger contribution from the direct LoRA path ($1-g \approx 0.88$), while allowing the recurrent contribution to be learned during training.

4.4 LoopedLoRA-Light: Token-Adaptive Inference

Standard LoopedLoRA evaluates all k recurrent steps for every token during inference. To reduce this overhead, we introduce **LoopedLoRA-Light**, which reuses the learned blending score $g(z_0)$ as an inference-time gating signal. Given a threshold $\tau \in [0, 1)$,

$$z_{\text{out}} = \begin{cases} z_0 + g(z_0)(z_k - z_0), & g(z_0) \geq \tau, \\ z_0, & g(z_0) < \tau. \end{cases} \tag{9}$$

When $g(z_0) \geq \tau$, LoopedLoRA-Light executes the recurrent path as in standard LoopedLoRA. Otherwise, it bypasses the recurrent computation and directly recovers the standard LoRA path:

$$h = W_0 x + \frac{\alpha}{r} B z_0 = W_0 x + \frac{\alpha}{r} BAx. \tag{10}$$

The same latent blender therefore serves two roles: continuous blending during training and hard execution gating during inference. Setting $\tau = 0$ recovers standard LoopedLoRA, while increasing τ bypasses recurrence for more tokens. This provides a controllable accuracy–latency trade-off without additional training or routing parameters.

4.5 Parameter and Computational Complexity

For each adapted projection, standard LoRA introduces $r(d_{\text{in}} + d_{\text{out}})$ trainable parameters. LoopedLoRA additionally introduces the recurrent matrix R, RMSNorm gain γ, and blender parameters (w, b), resulting in

$$\Delta_{\text{param}} = r^2 + 2r + 1 = (r + 1)^2. \tag{11}$$

The additional parameter count is independent of the backbone dimensions and loop depth k. At $r = 16$, LoopedLoRA adds only 289 parameters per adapted projection. Across all adapted projections in Qwen3-4B-Base, this corresponds to a 0.22% increase over standard LoRA.

The recurrent operator requires approximately $2kr^2 + \mathcal{O}(kr)$ FLOPs per token and adapted projection. For $r = 16$ and $k = 4$, the recurrent matrix multiplications require only $2{,}048$ FLOPs. Despite this small arithmetic cost, repeated execution of small recurrent operations can incur non-negligible kernel-launch and memory-access overhead. We address this practical overhead through the fusion-friendly operator design above and evaluate its inference efficiency in Section 5.5.

5 Experiments

5.1 Experimental Setup

Backbones and Tasks. We evaluate LoopedLoRA primarily on Qwen3-4B-Base and Qwen3-8B-Base (Yang et al., 2025) across three domains. For mathematical reasoning, models are fine-tuned on MetaMathQA (Yu et al., 2024) (395K examples) and evaluated on GSM8K-CoT (Cobbe et al., 2021), SVAMP (Patel et al., 2021), and MATH-500 (Hendrycks et al., 2021). For code generation, models are fine-tuned on CodeFeedback-Python (Zheng et al., 2024) (105K examples) and

Table 1: Zero-shot performance (%) on mathematical reasoning and code generation. All methods use rank $r = 16$ unless specified. Bold denotes the best reported result within each backbone.

Method	Params	GSM8K	SVAMP	MATH-500	HumanEval	HumanEval+
Qwen3-4B-Base						
LoRA ($r = 16$)	33.0M	76.48	76.06	48.80	82.32	**74.39**
LoRA ($r = 32$)	66.1M	76.57	80.67	48.60	79.27	69.51
DoRA ($r = 16$)	34.1M	79.30	81.33	50.20	81.10	73.17
Stable-LoRA ($r = 16$)	33.0M	82.11	80.33	50.40	78.66	71.95
AuroRA ($r = 16$)	33.1M	62.47	61.67	49.20	76.22	68.29
PiSSA ($r = 16$)	33.0M	72.93	75.00	46.60	76.83	71.34
HiRA ($r = 16$)	33.0M	48.07	52.33	50.00	71.95	64.63
MiSS ($r = 16$)	17.7M	71.27	74.00	41.60	75.00	70.73
LoopedLoRA	33.1M	**89.44**	**89.83**	**52.00**	**83.83**	74.12
Qwen3-8B-Base						
LoRA ($r = 16$)	43.6M	73.62	72.67	52.60	81.98	73.66
LoRA ($r = 32$)	87.3M	74.80	76.00	53.20	81.93	**74.39**
DoRA ($r = 16$)	45.0M	71.80	66.67	50.80	81.10	73.78
Stable-LoRA ($r = 16$)	43.6M	78.17	80.00	53.40	82.04	74.00
AuroRA ($r = 16$)	43.7M	34.95	38.00	52.20	81.71	73.17
PiSSA ($r = 16$)	43.6M	73.09	75.33	47.40	78.66	73.17
HiRA ($r = 16$)	43.6M	67.48	68.00	53.40	81.93	74.00
MiSS ($r = 16$)	22.4M	73.69	77.00	45.00	80.10	73.39
LoopedLoRA	43.7M	**86.58**	**87.33**	**55.80**	**82.32**	**74.39**

evaluated on HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al., 2023). For commonsense reasoning, models are fine-tuned on commonsense_170k (Hu et al., 2023) and evaluated on BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2020), ARC-Easy and ARC-Challenge (Clark et al., 2018), and OpenBookQA (Mihaylov et al., 2018). We further evaluate cross-architecture transfer on Llama-3.2-3B and Llama-3.1-8B (Meta, 2024; Grattafiori et al., 2024); these results are reported in Appendix B.

Training and Evaluation. All methods are trained for two epochs with identical data exposure using DeepSpeed ZeRO-2 in bfloat16. We use AdamW with a learning rate of 2×10^{-4} and cosine decay, an effective batch size of 32, and a maximum sequence length of 2,048. Adapters are applied to all seven linear projections (`q, k, v, o, gate, up, down`). Unless otherwise specified, all methods use rank $r = 16$. Commonsense benchmarks follow the evaluation protocol of DoRA (Liu et al., 2024), while mathematical and code benchmarks are evaluated using `lm-evaluation-harness` (Gao et al., 2023). Further implementation details are provided in Appendix A.

Baselines. We compare against standard LoRA (Hu et al., 2022) at $r \in \{16, 32\}$, DoRA (Liu et al., 2024), Stable-LoRA (Wu et al., 2026), the nonlinear adapter AuroRA (Dong et al., 2025), PiSSA (Meng et al., 2024), HiRA (Huang et al., 2025), and MiSS (Kang & Yin, 2026). All baselines except Stable-LoRA use the official Hugging Face PEFT implementations. We reimplement Stable-LoRA following the original method description and evaluate it under the same protocol.

5.2 Mathematical Reasoning and Code Generation

Table 1 reports zero-shot performance on mathematical reasoning and code generation.

Mathematical Reasoning. LoopedLoRA achieves the highest accuracy across all three mathematical reasoning benchmarks on both backbones. On Qwen3-4B-Base, it improves over LoRA ($r = 16$) by 12.96 and 13.77 percentage points on GSM8K and SVAMP, reaching 89.44% and 89.83%, respectively; the corresponding gains on Qwen3-8B-Base are 12.96 and 14.66 points. It also achieves the highest MATH-500 accuracy, with 52.00% on 4B and 55.80% on 8B. Doubling the LoRA rank to $r = 32$ yields much smaller gains. On GSM8K, LoRA ($r = 32$) reaches 76.57% and 74.80% on the 4B and 8B models, compared with 89.44% and 86.58% for LoopedLoRA. These results suggest that increasing computational depth within the low-rank subspace is particularly beneficial for mathematical reasoning beyond simply widening the LoRA bottleneck.

Table 2: Zero-shot accuracy (%) on eight commonsense reasoning benchmarks. All methods use rank $r = 16$ unless specified. Bold denotes the best reported result within each backbone.

Method	Params	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-C	ARC-E	OBQA	Avg
Qwen3-4B-Base										
LoRA ($r = 16$)	33.0M	72.04	87.83	80.29	93.88	86.00	88.96	94.80	90.40	86.78
LoRA ($r = 32$)	66.1M	**74.46**	90.10	81.63	95.68	87.77	89.85	96.17	92.40	88.51
DoRA ($r = 16$)	34.1M	73.49	89.28	**81.78**	95.89	87.21	90.53	96.38	**93.20**	88.47
Stable-LoRA ($r = 16$)	33.0M	74.19	90.21	80.66	95.80	**87.85**	**91.21**	**96.51**	91.60	88.50
AuroRA ($r = 16$)	33.1M	74.23	89.39	80.58	95.71	87.07	90.36	95.76	92.20	88.16
PiSSA ($r = 16$)	33.0M	68.41	83.79	78.81	91.11	83.50	82.17	92.42	85.40	83.20
HiRA ($r = 16$)	33.0M	70.52	88.47	80.40	94.27	82.00	88.91	96.25	90.20	86.38
MiSS ($r = 16$)	17.7M	62.17	49.29	32.50	25.08	47.59	22.70	25.08	27.60	36.50
LoopedLoRA	33.1M	74.34	**90.42**	81.06	**95.90**	87.21	90.70	**96.51**	93.00	**88.64**
Qwen3-8B-Base										
LoRA ($r = 16$)	43.6M	75.78	91.11	83.01	95.82	88.37	92.92	97.73	93.40	89.77
LoRA ($r = 32$)	87.3M	**77.03**	90.75	82.60	96.74	88.29	92.58	97.22	92.80	89.75
DoRA ($r = 16$)	45.0M	75.84	91.57	**83.32**	96.71	88.61	92.49	97.77	93.60	89.99
Stable-LoRA ($r = 16$)	43.6M	76.70	91.35	83.11	96.45	**89.16**	92.66	97.69	92.80	89.99
AuroRA ($r = 16$)	43.7M	75.72	91.68	82.73	**96.90**	88.50	**93.17**	97.73	92.00	89.80
PiSSA ($r = 16$)	43.6M	70.70	86.07	82.14	93.86	86.82	86.77	94.49	88.40	86.16
HiRA ($r = 16$)	43.6M	73.06	90.59	82.80	95.27	85.79	92.58	97.73	90.60	88.55
MiSS ($r = 16$)	22.4M	69.54	85.64	80.19	92.04	83.27	82.68	93.10	85.60	84.01
LoopedLoRA	43.7M	76.15	**91.78**	82.91	96.76	88.71	**93.17**	**98.02**	**93.80**	**90.16**

Table 3: Design ablation on Qwen3-4B-Base. The default model uses $k = 4$, learned latent blending, RMSNorm, and SiLU. Avg denotes the unweighted average across the three domains.

Aspect	Variant	CS	Math	Code	Avg
Depth	$k = 0$ (LoRA)	86.78	67.11	78.36	77.42
	$k = 1$	87.35	70.50	78.36	78.74
	$k = 2$	87.93	73.50	78.40	79.94
	$k = 3$	88.21	76.00	78.55	80.92
	$k = 4$	**88.64**	77.09	78.98	**81.57**
	$k = 5$	**88.64**	**77.10**	78.90	81.55
	$k = 6$	88.54	76.80	**78.99**	81.44
Blend	$g = 1$ (recurrent only)	87.04	73.60	74.20	78.28
	Fixed $g \approx 0.119$	88.06	75.30	75.80	79.72
	Learned $g(z_0)$ (Ours)	**88.64**	**77.09**	**78.98**	**81.57**
Operator	w/o RMSNorm	88.46	74.60	74.60	79.22
	w/o SiLU	88.58	72.60	72.50	77.89
	RMSNorm + SiLU (Ours)	**88.64**	**77.09**	**78.98**	**81.57**

Code Generation. On HumanEval, LoopedLoRA obtains 83.83% on Qwen3-4B-Base and 82.32% on Qwen3-8B-Base, outperforming all compared rank-16 baselines. On HumanEval+, LoopedLoRA remains competitive, achieving 74.12% and 74.39% on the two backbones. The gains are smaller than on mathematical reasoning, but indicate that the larger improvements in math are obtained while maintaining competitive code-generation performance.

5.3 Commonsense Reasoning

We next evaluate zero-shot performance across eight commonsense reasoning benchmarks. Table 2 reports individual task accuracy and the average across tasks. LoopedLoRA achieves the highest average accuracy on both backbones, reaching 88.64% on Qwen3-4B-Base and 90.16% on Qwen3-8B-Base, which improve over LoRA ($r = 16$) by 1.86 and 0.39 percentage points, respectively. It also slightly exceeds parameter-doubled LoRA ($r = 32$) while using approximately half as many trainable parameters (88.64% versus 88.51% on 4B; 90.16% versus 89.75% on 8B), indicating that simply increasing the LoRA rank does not match the performance of recurrent subspace depth in these settings.

5.4 Ablation Studies

We ablate the main architectural choices of LoopedLoRA on Qwen3-4B-Base. To summarize behavior across domains, we report the average over eight commonsense tasks (**CS**), three mathematical reasoning tasks (**Math**), and two code-generation tasks (**Code**). All variants use $r = 16$ and are trained for two epochs under the same protocol. Unless otherwise stated, each variant changes only the named design choice and retains the remaining default components.

Effect of Recurrent Depth. Performance improves steadily as recurrent depth increases from $k = 0$ to $k = 4$, particularly on mathematical reasoning, where the average increases from 67.11 to 77.09. A single nonlinear transformation ($k = 1$) improves over standard LoRA, but remains 6.59 percentage points below $k = 4$ on Math. Performance largely saturates around $k = 4$–5, motivating $k = 4$ as our default. These results support the benefit of repeated refinement beyond a single recurrent refinement step.

Latent Blending. Using only the recurrent output ($g = 1$) performs substantially worse than blending it with the direct LoRA path. A fixed LoRA-biased blend improves performance, but remains below the learned blender. Compared with fixed $g \approx 0.119$, the learned $g(z_0)$ improves the Math and Code averages by 1.79 and 3.18 points, respectively. Thus, learning the blending function provides an additional benefit over the fixed residual mixture used in this ablation.

Recurrent Operator. Removing RMSNorm reduces the Math and Code averages by 2.49 and 4.38 points, while replacing SiLU with an identity transformation results in larger drops of 4.49 and 6.48 points. These results support both normalization and nonlinearity as important components of the recurrent operator.

Table 4: Training and inference efficiency on Qwen3-4B-Base. Train Thpt. denotes training throughput in samples/s, Prefill is measured in ms/request, Decode in ms/token, and Mem. denotes peak GPU memory in GB. Δ Dec. is relative to LoRA ($r = 16$). †LoopedLoRA-Light reuses the full LoopedLoRA checkpoint and requires no additional training.

Method	Params	Train Time	Train Thpt.	Prefill	Decode	Δ Dec.	Mem.
LoRA ($r = 16$)	33.03M	5:05:53	18.57	46.2	14.04	–	7.86
LoRA ($r = 32$)	66.06M	5:03:41	18.71	45.0	14.15	+0.8%	8.02
LoopedLoRA	33.10M	6:31:35	14.51	61.9	13.51	**−3.8%**	7.86
LoopedLoRA-Light	33.10M	–†	–	62.8	**13.17**	**−6.2%**	7.89

5.5 Training and Inference Efficiency

We evaluate the training and inference efficiency of LoopedLoRA on Qwen3-4B-Base. Training efficiency is measured on Commonsense170k for two epochs. For inference, experiments use bfloat16 on a single GPU with batch size 1 and a commonsense workload of 128 samples with an average input length of 106 tokens and eight decoded tokens per sample. Prefill is measured in eager mode, while autoregressive decoding uses CUDA graph replay. LoopedLoRA uses the fused Triton implementation described in Section 4.2.

Training and Inference Efficiency. LoopedLoRA adds only 0.22% more trainable parameters than LoRA ($r = 16$), while increasing wall-clock training time by 27.9% and reducing training throughput from 18.57 to 14.51 samples/s. At inference, LoopedLoRA incurs higher prefill latency than LoRA ($r = 16$) (61.9 versus 46.2 ms), while its decode latency remains comparable under CUDA graph replay (13.51 versus 14.04 ms/token) with the same peak memory usage. Reusing the full LoopedLoRA checkpoint, LoopedLoRA-Light further reduces decode latency to 13.17 ms/token at $\tau = 0.1$, without additional training or parameters.

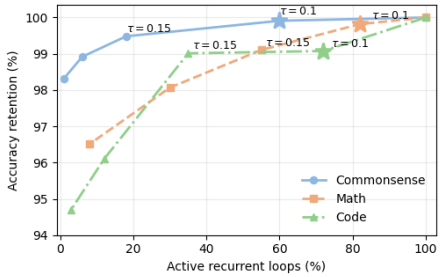
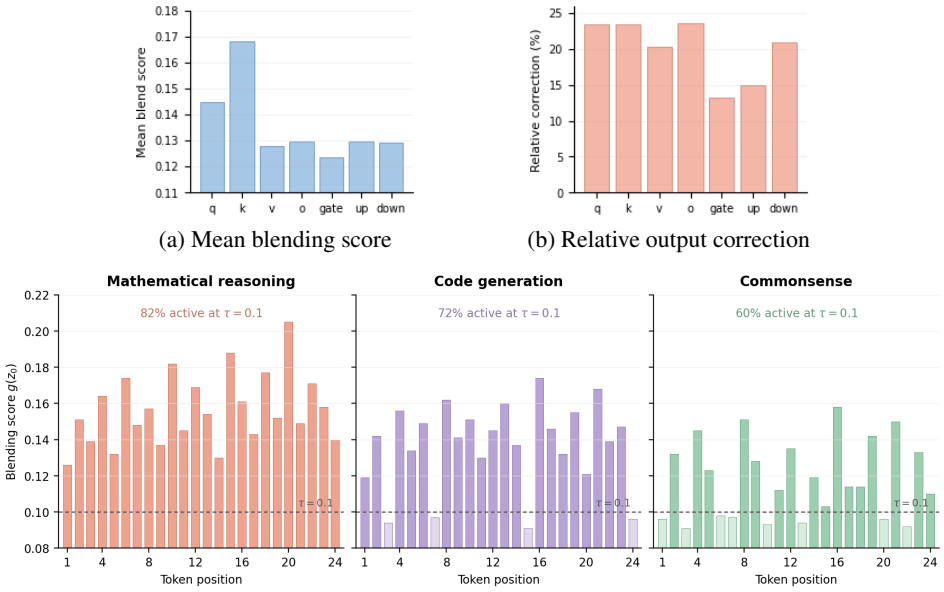


Figure 2: **Accuracy–compute trade-off of LoopedLoRA-Light.** Accuracy retention is normalized by full LoopedLoRA performance ($\tau = 0$). Stars indicate the recommended operating points at $\tau = 0.1$.

Accuracy–Compute Trade-off. LoopedLoRA-Light uses the learned blending score as an inference-time gate, allowing recurrent computation to be selectively bypassed by adjusting a single threshold τ. Figure 2 shows the trade-off between the fraction of active recurrent adapter evaluations and accuracy across the three evaluation domains. We use $\tau = 0.1$ as the recommended operating point, where only 60%, 82%, and 72% of recurrent adapter evaluations remain active for commonsense, math, and code, respectively, while retaining 99.91%, 99.82%, and 99.08% of full LoopedLoRA accuracy. Larger



(a) Mean blending score

(b) Relative output correction

(c) Token-wise blending scores across domains

Figure 3: **Analysis of learned recurrent refinement.** (a) Average blending score across projection types. (b) Relative correction of the adapter output induced by recurrent refinement. (c) Token-wise blending scores for randomly selected sequences from mathematical reasoning, code generation, and commonsense reasoning. The dashed line denotes the LoopedLoRA-Light threshold $\tau = 0.1$; annotations report the domain-level fraction of adapter-token evaluations for which recurrent refinement remains active.

thresholds further reduce recurrent computation with a gradual decrease in accuracy, allowing different accuracy–compute operating points from a single checkpoint without retraining. Detailed threshold-wise results are provided in Appendix C.

5.6 Analysis of Learned Recurrent Refinement

We analyze learned recurrent refinement across projections, network depth, and evaluation domains. Figure 3(a) shows that the mean blending score is relatively small ($\mathbb{E}[g] = 0.136$) but varies across projections, peaking in the key projection. To quantify its effect beyond the gate value, we measure the relative output correction $\frac{\|B(z_{\text{out}} - z_0)\|}{\|Bz_0\| + \epsilon}$. The blended recurrent path changes the adapter output by about 20% on average, with larger corrections in attention projections (Figure 3(b)). Although the mean blending score increases from 0.127 in early layers to 0.145 in late layers, correction is largest in early layers (29.2%), reflecting its dependence on the gate, recurrent update, and decoder. Figure 3(c) shows that blending scores vary across tokens and domains. At $\tau = 0.1$, recurrence remains active for 60%, 82%, and 72% of adapter-token evaluations on commonsense, mathematical reasoning, and code-generation tasks, respectively. Thus, the learned blender modulates recurrent refinement across tokens and domains. Additional latent-state analyses are provided in Appendix D.

6 Conclusion and Limitations

We introduced **LoopedLoRA**, a parameter-efficient adaptation method that adds recurrent processing within the low-rank adapter subspace. Its latent blender combines direct and recurrent paths, while LoopedLoRA-Light enables token-adaptive inference without additional training. Across mathematical reasoning, code generation, and commonsense reasoning, LoopedLoRA achieves its largest gains on mathematical reasoning while remaining competitive elsewhere with negligible parameter overhead. Unlike standard linear LoRA, its nonlinear, input-dependent adapter is not generally mergeable and incurs additional training and prefill cost. Overall, recurrent subspace depth offers an effective alternative to increasing adapter rank.

AI USE STATEMENT

We used generative AI tools to assist with writing and debugging implementation code. We did not use generative AI tools to generate or analyze experimental results, create figures, or make scientific claims. All AI-assisted text and code were reviewed and verified by the authors, who take full responsibility for the final content of this work.

ETHICS STATEMENT

This work studies a parameter-efficient adaptation method for language models and does not involve human subjects or the collection of private data. LoopedLoRA does not introduce new data sources or intended deployment settings beyond those of the underlying models and fine-tuning datasets.

REPRODUCIBILITY STATEMENT

Section 4 specifies the LoopedLoRA architecture, recurrent update, initialization, and computational complexity. Section 5 describes the training data, adapted modules, optimization settings, and evaluation benchmarks. Appendix A provides additional training, evaluation, prompting, decoding, and inference-benchmark details. We evaluate all adapter baselines using the same data exposure and evaluation protocol.

REFERENCES

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of AAAI*, volume 34, pp. 7432–7439, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Christopher Clark, Matt Gardner, Iacopo Alfano, et al. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of NAACL-HLT*, pp. 2924–2936, 2019.

Peter Clark, Isaac Cowhey, Oren Etzioni, et al. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations (ICLR)*, 2019.

Haonan Dong, Wenhao Zhu, Guojie Song, and Liang Wang. AuroRA: Breaking low-rank bottleneck of LoRA with nonlinear mapping. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Jiayi Lin, et al. A framework for few-shot language model evaluation, 2023.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning (ICML)*, 2023.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034, 2015. doi: 10.1109/ICCV.2015.123.

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 *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

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 (ICLR)*, 2022.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Lee. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 5254–5276, 2023.

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

Jiale Kang and Qingyu Yin. Miss: Revisiting the trade-off in lora with an efficient shard-sharing structure. In *International Conference on Learning Representations*, 2026.

Nanbeige Lab, Chen Yang, Chengrui Huang, Fufeng Lan, Hanhui Chen, Hao Zhou, Huatong Song, Jiaqi Cao, Jiaying Zhu, Jinlin Niu, et al. Nanbeige4. 2-3b: Unlocking agentic capabilities in a compact model. *arXiv preprint arXiv:2607.22083*, 2026.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *International Conference on Learning Representations (ICLR)*, 2020.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 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 (ICML)*, 2024.

Xu Luo, Yongbin Liu, Chunping Ouyang, and Ying Yu. RanLoRA: Residual-aware nonlinear low-rank adaptation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17243–17258, 2026.

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 (NeurIPS)*, 2024.

Meta. Llama 3.2 model card. `https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md`, 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 EMNLP*, pp. 2381–2391, 2018.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094, 2021.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. In *Proceedings of AAAI*, volume 34, pp. 8732–8740, 2020.

Maarten Sap, Hannah Rashkin, Derek Chen, et al. Social iqa: Commonsense reasoning about social interactions. In *Proceedings of EMNLP-IJCNLP*, pp. 4463–4473, 2019.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Yize Wu, Ke Gao, Ling Li, and Yanjun Wu. Stable-LoRA: Stabilizing feature learning of low-rank adaptation. In *International Conference on Learning Representations (ICLR)*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Liu Yang, Kangwook Lee, Robert D Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International Conference on Learning Representations (ICLR)*, 2024.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. MetaMath: Bootstrap your own mathematical questions for large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Abbas Zeitoun, Lucas Torroba-Hennigen, and Yoon Kim. Hyperloop transformers. *arXiv preprint arXiv:2604.21254*, 2026.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, et al. Hellaswag: Can a machine really finish your sentence? In *Proceedings of ACL*, pp. 4791–4800, 2019.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

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.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

A Experimental Details

A.1 Training and Evaluation Configuration

We train all Qwen and Llama models using four NVIDIA H100 GPUs. Mathematical reasoning models are trained on MetaMathQA, code-generation models on CodeFeedback-Python, and commonsense-reasoning models on `commonsense_170k`. All methods are trained for two epochs with identical data exposure using DeepSpeed ZeRO-2 in bfloat16. We use AdamW with a learning rate of 2×10^{-4} and cosine decay, an effective batch size of 32, and a maximum sequence length of 2,048. Adapters are applied to the `q`, `k`, `v`, `o`, `gate`, `up`, and `down` projections. Unless otherwise specified, adapters use $r = 16$ and $\alpha = 16$.

All methods are evaluated with the same benchmark splits, zero-shot settings, decoding configuration, stop sequences, answer-extraction rules, and metrics. We use bfloat16 with `model.eval()` and left padding. Decoding is greedy (`do_sample=False`), with a maximum generation length of 256 tokens for mathematical reasoning and 1,024 tokens for HumanEval and HumanEval+. We evaluate the checkpoint obtained after two training epochs.

For Qwen models, we place the benchmark-provided math or code prompt in the user-message content and apply the native chat template with `add_generation_prompt=True`. We disable Qwen3 thinking mode (`enable_thinking=False`) to match training and add `<|im_end|>` as an end-of-sequence token. For Llama models, we likewise apply the corresponding chat template to the benchmark prompt. We use the Llama-3 Instruct template for Llama-3.1-8B-Base and the following fallback template for Llama-3.2-3B-Base:

```
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

{benchmark prompt}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

We add `<|eot_id|>` as an end-of-sequence token for Llama models.

For commonsense reasoning, we follow the DoRA evaluation protocol and do not apply a chat template. All models use the same Alpaca-style prompt, consistent with commonsense fine-tuning:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:

{instruction}

### Response:
```

We use greedy decoding with `max_new_tokens=16` and the same task-specific label-extraction rules across methods.

Reproducibility across random seeds. We additionally repeated the main experiments across all three evaluation domains—mathematical reasoning, code generation, and commonsense reasoning—with three random seeds. The resulting performance was highly consistent across runs, with only minor variation from the reported results, indicating that the observed gains of Looped-LoRA are consistent across random seeds.

A.2 Inference Efficiency Benchmark Protocol

We evaluate inference efficiency on Qwen3-4B-Base in bfloat16 using a single NVIDIA H100 GPU and batch size 1. The commonsense workload contains 128 samples with an average input length of 106 tokens, followed by eight autoregressively decoded tokens. Prefill latency is measured in eager mode, while decode latency is measured with CUDA graph replay.

All methods are evaluated using the same input samples, prompt and decode lengths, GPU and software environment, timing boundaries, and peak-memory measurement procedure. Model loading

Table 5: Zero-shot performance (%) on mathematical reasoning and code generation with Llama backbones. All methods use rank $r = 16$ unless specified. Bold denotes the best result within each backbone.

Method	Params	GSM8K	SVAMP	MATH-500	HumanEval	HumanEval+
Llama-3.2-3B-Base						
LoRA ($r = 16$)	24.31M	66.25	70.50	19.40	41.46	35.98
LoRA ($r = 32$)	48.63M	67.99	71.33	19.60	40.24	37.20
DoRA ($r = 16$)	25.09M	65.58	70.17	20.40	40.24	33.54
Stable-LoRA ($r = 16$)	24.31M	65.87	70.83	19.20	41.46	36.59
AuroRA ($r = 16$)	24.39M	67.00	72.00	20.50	39.36	35.88
PiSSA ($r = 16$)	24.31M	66.40	70.83	21.00	41.07	37.41
HiRA ($r = 16$)	24.31M	53.75	61.33	11.60	32.32	28.66
MiSS ($r = 16$)	12.39M	59.97	62.33	14.80	37.80	36.59
LoopedLoRA	24.37M	**68.40**	**72.67**	**22.40**	**42.07**	**38.40**
Llama-3.1-8B-Base						
LoRA ($r = 16$)	41.94M	71.01	72.33	29.00	52.27	49.39
LoRA ($r = 32$)	83.89M	73.16	73.33	30.00	**54.88**	49.39
DoRA ($r = 16$)	43.32M	74.00	74.00	30.00	52.44	50.00
Stable-LoRA ($r = 16$)	41.94M	66.79	73.67	29.40	49.39	46.95
AuroRA ($r = 16$)	42.03M	75.50	76.50	30.50	53.39	51.12
PiSSA ($r = 16$)	41.94M	63.99	70.00	27.60	52.66	50.00
HiRA ($r = 16$)	41.94M	41.02	55.00	25.20	48.17	43.90
MiSS ($r = 16$)	22.02M	64.52	64.00	22.60	48.78	45.12
LoopedLoRA	42.01M	**79.15**	**81.00**	**31.60**	54.66	**51.61**

time is excluded from latency measurements. Routing statistics collection is disabled during timing (`LOOPED_SKIP_STATS=0`). Decode latency is reproduced across three independent runs, with run-to-run variation below 0.01 ms/token.

The internal kernel implementations are not identical across methods. LoopedLoRA uses the fused Triton implementation described in Section 4.2, which combines RMSNorm, the recurrent latent updates, SiLU, and latent blending within the adapted projection. LoopedLoRA-Light additionally performs the gate-threshold decision and inactive loop skipping within the fused kernel.

Accordingly, the reported latency should be interpreted as an end-to-end comparison of optimized serving implementations under the same workload and execution protocol, rather than as a microbenchmark of implementations with identical kernel structures. The lower decode latency of LoopedLoRA therefore does not indicate lower arithmetic complexity than standard LoRA, but reflects the effects of kernel fusion and CUDA graph replay in this serving setting.

LoopedLoRA-Light uses the same checkpoint as full LoopedLoRA and applies token-level loop skipping at inference time with $\tau = 0.1$; it introduces no additional training or parameters.

B Additional Results on Llama Models

B.1 Mathematical Reasoning and Code Generation

Table 5 reports zero-shot performance on mathematical reasoning and code generation for Llama-3.2-3B-Base and Llama-3.1-8B-Base.

On mathematical reasoning, LoopedLoRA achieves the best performance on all three benchmarks for both Llama backbones. Compared with LoRA ($r = 16$), it improves GSM8K, SVAMP, and MATH-500 by 2.15, 2.17, and 3.00 percentage points, respectively, on Llama-3.2-3B-Base, and by 8.14, 8.67, and 2.60 percentage points on Llama-3.1-8B-Base. On code generation, LoopedLoRA achieves the best scores on both HumanEval benchmarks for Llama-3.2-3B-Base. On Llama-3.1-8B-Base, it obtains the best HumanEval+ score and remains competitive on HumanEval, where LoRA ($r = 32$) achieves the highest score.

Table 6: Zero-shot accuracy (%) on eight commonsense reasoning benchmarks with Llama backbones. All methods use rank $r = 16$ unless specified. Bold denotes the best result within each backbone.

Method	Params	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-C	ARC-E	OBQA	Avg
Llama-3.2-3B-Base										
LoRA ($r = 16$)	24.31M	71.83	87.27	80.19	95.03	86.50	78.41	89.98	82.80	84.00
LoRA ($r = 32$)	48.63M	73.09	87.65	80.60	94.83	86.42	78.41	89.44	85.60	84.51
DoRA ($r = 16$)	25.09M	**73.46**	87.05	**81.01**	**95.30**	86.27	78.50	90.40	83.80	84.47
Stable-LoRA ($r = 16$)	24.31M	73.39	87.38	80.60	94.86	**87.85**	**78.84**	89.81	84.20	84.62
AuroRA ($r = 16$)	24.39M	72.62	**87.83**	80.48	94.86	86.87	77.81	90.40	84.60	84.43
PiSSA ($r = 16$)	24.31M	68.56	83.90	78.66	90.80	82.95	71.59	85.23	81.00	80.34
HiRA ($r = 16$)	24.31M	66.48	83.57	75.64	91.02	76.40	73.12	85.52	74.40	78.27
MiSS ($r = 16$)	12.39M	61.74	62.73	67.55	51.25	63.77	43.94	53.96	54.60	57.44
LoopedLoRA	24.37M	72.91	87.81	80.96	95.25	86.58	78.67	**90.49**	**86.40**	**84.88**
Llama-3.1-8B-Base										
LoRA ($r = 16$)	41.94M	74.50	**90.42**	82.91	95.58	88.66	83.30	93.27	89.00	87.20
LoRA ($r = 32$)	83.89M	75.23	89.77	82.80	96.09	88.40	82.94	92.38	87.20	86.85
DoRA ($r = 16$)	43.32M	75.35	90.24	82.60	96.08	89.11	83.96	93.73	**90.20**	87.66
Stable-LoRA ($r = 16$)	41.94M	75.10	90.32	82.09	96.51	89.03	83.32	93.56	88.60	87.32
AuroRA ($r = 16$)	42.03M	74.44	90.39	83.03	96.66	**89.79**	83.55	93.27	87.60	87.34
PiSSA ($r = 16$)	41.94M	71.19	85.85	80.50	93.14	84.45	76.96	89.27	84.20	83.19
HiRA ($r = 16$)	41.94M	71.93	89.01	79.32	94.62	85.32	80.89	92.34	82.60	84.50
MiSS ($r = 16$)	22.02M	62.17	50.60	33.11	25.00	51.54	22.78	25.00	27.60	37.22
LoopedLoRA	42.01M	**75.50**	90.37	**83.16**	**96.74**	89.59	**84.22**	**94.07**	**90.20**	**87.98**

Table 7: Detailed LoopedLoRA-Light trade-offs on Qwen3-4B-Base. Active denotes the fraction (%) of adapter-token evaluations that execute recurrent refinement; Acc. denotes task accuracy (%).

	Commonsense		Math		Code	
τ	Active	Acc.	Active	Acc.	Active	Acc.
0.00	100	88.64	100	77.09	100	78.98
0.10	60	88.56	82	76.95	72	78.25
0.15	18	88.18	55	76.40	35	78.20
0.20	6	87.68	30	75.60	12	75.90
0.30	1	87.15	8	74.40	3	74.80

B.2 Commonsense Reasoning

Table 6 reports zero-shot accuracy across eight commonsense reasoning benchmarks for the two Llama backbones.

LoopedLoRA achieves the highest average accuracy on both Llama backbones, reaching 84.88% on Llama-3.2-3B-Base and 87.98% on Llama-3.1-8B-Base. Compared with LoRA ($r = 16$), this corresponds to gains of 0.88 and 0.78 percentage points, respectively. It also exceeds the strongest non-Looped baseline by 0.26 percentage points on Llama-3.2-3B-Base and 0.32 percentage points on Llama-3.1-8B-Base. These results show that the gains also extend to the two evaluated Llama backbones.

C Detailed LoopedLoRA-Light Trade-offs

Table 7 reports the accuracy–compute trade-off of LoopedLoRA-Light across thresholds. The active ratio denotes the fraction of adapter-token evaluations that execute recurrent refinement. Setting $\tau = 0$ recovers full LoopedLoRA. At $\tau = 0.1$, LoopedLoRA-Light reduces recurrent evaluations by 40%, 18%, and 28% on commonsense, mathematical reasoning, and code generation, respectively, while retaining near-full accuracy across all three domains. Larger thresholds further reduce recurrent computation with a gradual decrease in accuracy.

D Additional Analysis of Recurrent Refinement

D.1 Latent Trajectories Across Recurrent Steps

We analyze how the recurrent latent state evolves across steps using the default LoopedLoRA checkpoint on Qwen3-4B-Base. We record latent states from 32 commonsense prompts and aggregate

statistics over tokens and adapted projections. For each recurrent step t, we measure the relative update $\frac{\|z_t - z_{t-1}\|}{\|z_0\| + \epsilon}$, the cosine similarity between z_t and z_0, and the root-mean-square (RMS) of z_t.

Table 8: Latent trajectories across recurrent steps on Qwen3-4B-Base. Relative update is normalized by the initial latent state z_0.

Step	**Relative update**	**Cosine to z_0**	**RMS**
1	1.7463	0.0200	0.3104
2	1.6117	0.0337	0.3126
3	1.5349	-0.0040	0.3201
4	1.5096	0.0297	0.3209

The relative update remains substantial across recurrent steps, while the cosine similarity to z_0 remains close to zero. Thus, recurrence does not merely rescale the direct latent state, but produces latent states with substantially different directions. The state RMS remains within a narrow range under the default configuration, indicating a consistent latent scale across recurrent steps.