LoGRA: Scaling LLM Reinforcement Learning with Low-Rank Gradient Sketches

Anonymous authors
Paper under double-blind review

Abstract

Reinforcement learning (RL) has greatly advanced the capabilities of large language models (LLMs), but its memory demands remain a barrier to broader adoption. We introduce LoGRA, an approach to RL post-training that reduces memory by retaining useful learning signals in low-rank gradient sketches. These compact representations support both model updates and efficient policy synchronization. To prevent overly large updates from disrupting learning, we complement gradient compression with predicted-KL step control, which estimates policy changes before applying each update and adjusts its magnitude accordingly. Across reasoning tasks, LoGRA reduces average training memory by up to 45.7% without sacrificing performance. It also enables stable training of a 27B-parameter model for over 1,100 steps on a single eight-GPU node, where dense Adam runs out of memory, making previously memory-infeasible RL training practical.

1 Introduction

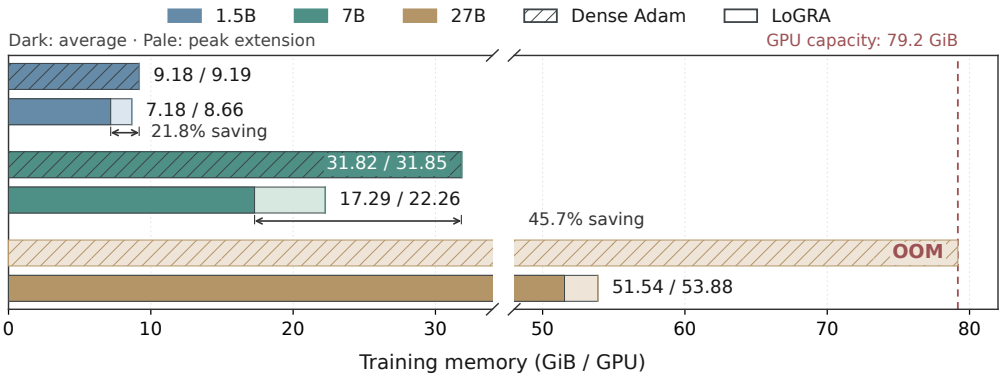


Figure 1: **Lower training memory across model scales.** Colors denote model sizes; hatching distinguishes Dense Adam from LoGRA. Dark bars show average memory and pale extensions reach peak memory (Table 1); labels report average / peak in GiB per training GPU. Arrows show average-memory savings over Dense Adam. The broken axis omits 34–48 GiB. The 27B Dense Adam bar marks OOM at GPU capacity, not a measured value.

Reinforcement learning (RL) has emerged as a key driver of advances in LLM, enabling models to improve through feedback on their own generated responses (Sheng et al., 2024; nem, 2025; Xu et al., 2026; Zhang et al., 2026b). However, its substantial memory requirements remain a major barrier to broader adoption. This challenge extends well beyond storing model parameters, as gradients and optimizer states can consume several times more memory than the weights themselves. For example, the BF16 weights of a 7B-parameter Qwen model (Bai et al., 2023) require approximately 14 GB, while Adam's (Kingma & Ba, 2014) two FP32 moment buffers add another 56 GB, even before accounting for gradients and activations. Consequently, hardware capable of running a model may still be insufficient to improve it through RL. Reducing this memory overhead is therefore critical to making RL post-training accessible to the broader research community.

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

To overcome this bottleneck, existing approaches span both system-level optimizations and memory-efficient training algorithms. At the system level, FSDP (Zhao et al., 2023) shards training states across GPUs, gradient checkpointing (Chen et al., 2016) reduces activation storage through recomputation, and CPU offloading moves selected training states to host memory (Rajbhandari et al., 2020). Parameter-efficient methods reduce the number of parameters that require gradients and optimizer states: LoRA (Hu et al., 2021), for example, freezes pretrained weights and learns small low-rank adapters. Gradient compression offers another route (Zhao et al., 2024; Su et al., 2025; Hao et al., 2024): they keep the original weight matrices trainable while using compact representations of gradients and optimizer states, thereby reducing training memory while still enabling effective updates to the full model. This perspective presents a new opportunity to reduce the memory requirements of RL post-training.

Motivated by this perspective, we introduce LoGRA, a method that uses low-rank gradient sketches to reduce the memory and communication costs of RL post-training, thereby lower the hardware barrier. Our intuition is that learning from rewards may require far less information than a full gradient can represent: with binary outcome rewards, each generated response receives only a single correctness signal, motivating a compact representation of the resulting learning signal (Schulman & Lab, 2025). LoGRA directly accumulates sketches during backpropagation, avoiding the construction and storage of full gradients for the selected weight matrices. These sketches produce updates that are merged into the model weights, with directions that can vary throughout training. The same compact representation is then reused to synchronize the rollout policy, extending the benefits of compression from training memory to communication.

Memory efficiency alone does not ensure stable learning: an overly large update can abruptly change the policy and disrupt previously learned behavior. We therefore introduce predicted-KL step control, which estimates how much a proposed update would change the model's next-token probabilities before applying it. Computed directly from the low-rank factors, this estimate guides the update scale to fit a prescribed KL budget.

We evaluate LoGRA across multiple reasoning tasks with models ranging from 1.5B to 27B parameters. LoGRA reduces average training memory by 21.8% at 1.5B and 45.7% at 7B without sacrificing task performance. These savings also make larger-scale training feasible: LoGRA sustains stable learning of a 27B model for over 1,100 steps on a single eight-GPU node, where the dense Adam baseline runs out of memory under the same hardware allocation. Ablations further clarify how projection design and KL budgets affect learning performance and training stability.

2 METHOD

An LLM contains large weight matrices, and storing a full gradient for each matrix adds substantial training memory. LoGRA replaces these full gradients with compact sketches and represents each weight update as the product of two thin matrices. We first describe this matrix representation, then explain its use in RL post-training and policy synchronization. Predicted KL provides an optional rule for choosing the update magnitude.

2.1 LOW-RANK GRADIENT SKETCHES

Full-gradient updates. Let θ denote the parameters of an LLM policy π_θ, and let $\mathcal{L}(\theta)$ be a differentiable RL training loss. Consider one weight matrix $W \in \mathbb{R}^{d\times k}$ in the model. Its gradient G has the same dimensions as W; for example, an SGD update takes the form

$$G = \nabla_W \mathcal{L} \in \mathbb{R}^{d\times k}, \qquad W \leftarrow W - \eta G, \tag{1}$$

where η is the learning rate. When a training batch is split into microbatches, standard training accumulates their contributions in a full $d \times k$ gradient buffer before updating W. This requires dk stored values in addition to the weights and any optimizer state. Our goal is to perform the update using a much smaller gradient representation.

A compact representation. Choose a rank $r \ll \min(d, k)$ and a random projection matrix $A \in \mathbb{R}^{r\times k}$. LoGRA represents the gradient by a sketch

$$S = GA^\top \in \mathbb{R}^{d\times r}. \tag{2}$$

Multiplication by $A^\top$ compresses the k columns of G into r columns of S. The sketch therefore stores dr values instead of dk. The projection A is sampled independently of the current batch and held fixed throughout gradient accumulation; it is not learned. By default, its entries are independent random signs scaled by $1/\sqrt{r}$.

Equation 2 defines the sketch mathematically, but does not require computing and then compressing a full gradient. Our implementation directly accumulates the projected gradient contributions from each microbatch into S in fp32. It does so by projecting the inputs used in the weight-gradient calculation before multiplying them by the backward derivatives. Thus, the compressed weights do not require a full gradient buffer, while the model's ordinary forward computation is preserved.

A low-rank weight update. To update W, we need a matrix with the same dimensions as W. The sketch S has only r columns, so we multiply it by A to obtain a $d \times k$ approximation to the full gradient:

$$\widehat{G} = SA \in \mathbb{R}^{d\times k}, \qquad W \leftarrow W - \eta SA. \tag{3}$$

In other words, we use SA in place of G in the standard gradient update. Although SA has the same dimensions as G, it is represented by two small matrices: S of size $d \times r$ and A of size $r \times k$. Because S has only r columns, every column of SA is a combination of these r columns; this is why the update has rank at most r. This compact representation approximates the full gradient rather than recovering it exactly. The resulting update is added directly to W, which remains a full-sized weight matrix.

An optimizer can also adjust the sketch before it is used in the update. We call this adjusted sketch U; it has the same dimensions as S, and $U = S$ for the basic SGD variant. The update then becomes

$$W \leftarrow W - \alpha\eta UA. \tag{4}$$

Here η sets the nominal step size, and α is an additional multiplier: $\alpha = 1$ leaves the step unchanged, while a smaller value reduces it. The predicted-KL controller described below chooses α without changing the shapes of U and A.

2.2 Applying LoGRA to RL post-training

We apply the sketch to attention and MLP projection matrices, holding other parameters fixed. Algorithm 1 summarizes the procedure, which supports differentiable RL losses. Each matrix has its own A, S, and U, while α is shared. Updates are merged directly into the weights, with no persistent adapter branch. Refreshing A each step allows successive updates to use different subspaces, so their sum need not have rank at most r. Here $\mathcal{L}_m$ denotes microbatch m's contribution to $\mathcal{L}$. The projected gradient in the algorithm is accumulated directly, without forming the full gradient matrix.

Algorithm 1 LoGRA with projection refreshing

Require: Policy π_θ, selected matrices W, rank r, step size η

Initialize rollout weights from θ.
for each training step **do**
  Sample responses from π_θ; construct $\mathcal{L}$. ▷ **1. Prepare**
  for each W **do**
    Sample $A \in \mathbb{R}^{r\times k}$ from a fresh seed.
    $S \leftarrow 0 \in \mathbb{R}^{d\times r}$
  for each microbatch m (θ and A fixed) **do** ▷ **2. Accumulate sketches**
    for each W **do**
      $S \leftarrow S + (\nabla_W \mathcal{L}_m)A^\top$ ▷ Direct sketch accumulation
  For each W: $U \leftarrow S$ (or use a sketch optimizer). ▷ **3. Choose the update**
  $\alpha \leftarrow 1$ ▷ Or use predicted KL; Section 2.3
  for each W **do** ▷ **4. Update and synchronize**
    $W \leftarrow W - \alpha\eta UA$ ▷ Trainer
    Send $\alpha\eta U$ and the seed for A to rollout.
    Regenerate A; apply the same update. ▷ Rollout
  Wait for synchronization to finish.

Memory and synchronization. The gradient accumulator shrinks from dk to dr values: for $d = k = 4096$ and $r = 64$, fp32 storage falls from 64 MiB to 1 MiB. Model weights, cached projections, activations, and other buffers still consume memory. Basic SGD requires no optimizer moments; stateful sketch optimizers add their own storage. For synchronization, the rollout engine regenerates A from its seed and applies the received factor $\alpha\eta U$. The payload is $O(dr)$ values plus a seed and metadata, instead of a full $d \times k$ update.

2.3 Step-size control with predicted KL

LoGRA specifies how to change the weights, but the size of that change also matters. A large update can abruptly change the model's predictions and destabilize training. We therefore estimate how much the proposed update would change the model's next-token probabilities, then scale the update to fit a prescribed budget. Let D denote the proposed update to all parameters; its component for each compressed weight matrix W is $D_W = -\eta U A$, using the factors and nominal step size defined above. We apply $\theta \leftarrow \theta + \alpha D$, where α controls the step size without changing its direction.

We select a small subset of responses from the current training batch and examine the predictions used to generate their tokens. For each prediction, the model sees s which is a prompt and the response generated so far. Let $\mathcal{P}$ collect these contexts. At each context, we compare the current model's probabilities over the vocabulary, $\pi_\theta(\cdot \mid s)$, with those of the proposed updated model, $\pi_{\theta+\alpha D}(\cdot \mid s)$. Both are evaluated on the same text context and at the same temperature $T > 0$: a higher temperature gives a flatter distribution.

KL divergence measures how much these two probability distributions differ. We average this change over the examined predictions as derived in detail in the the Appendix A:

$$\frac{1}{|\mathcal{P}|} \sum_{s \in \mathcal{P}} D_{\mathrm{KL}}(\pi_\theta(\cdot \mid s) \,\|\, \pi_{\theta+\alpha D}(\cdot \mid s)) \approx \alpha^2 q(D). \tag{5}$$

The left side is the average KL between the current and proposed policies over the examined contexts. On the right, $q(D)$ is our estimate of this average change for the unscaled proposal ($\alpha = 1$). The approximation predicts a quadratic dependence on step size: halving α reduces the predicted KL by a factor of four.

We specify a KL budget $\delta > 0$, the allowed average predicted change in next-token probabilities for one training step. A smaller budget keeps the updated policy closer to the current one. Equation 5 gives a way to control this change: rescale the proposed update by α so that $\alpha^2 q(D) \le \delta$. Thus, we first estimate the proposal's predicted KL $q(D)$, then choose its scale to fit the budget.

To estimate $q(D)$, we examine how the proposal would change the scores assigned to candidate next tokens. Let v index vocabulary tokens, $z_v(s)$ be the score (logit) of token v at context s, and $z(s)$ collect these scores. The vector $p(s) = \mathrm{softmax}(z(s)/T)$ contains their probabilities at temperature T. Let $\dot{z}_v(s)$ be the rate at which score $z_v(s)$ changes along D, evaluated at the current weights. The predicted KL is

$$q(D) = \frac{1}{2|\mathcal{P}|T^2} \sum_{s \in \mathcal{P}} \mathrm{Var}_{v \sim p(s)}[\dot{z}_v(s)]\,. \tag{6}$$

Derived could be found in the Appendix A. Here the variance measures how differently the token scores change, weighted by their current probabilities $p(s)$, and $|\mathcal{P}|$ is the number of examined contexts.

Rescale the update to fit the budget. Once $q(D)$ is available, the ratio $\delta/q(D)$ tells us how the budget compares with the proposed change. Because predicted KL scales with α^2, we take the square root of this ratio. With a chosen upper limit $\alpha_{\max} > 0$ on the update multiplier, we set

$$\alpha = \min\left\{\alpha_{\max}, \sqrt{\delta/q(D)}\right\}, \tag{7}$$

for finite positive $q(D)$. For example, a proposal with $q(D) = 4\delta$ is scaled to at most one half of its original size. The cap prevents excessive enlargement when the predicted KL is small. This rule controls the predicted change; the realized KL can differ because the estimate is a local approximation. A constant budget keeps δ fixed, while an annealed budget gradually lowers it to permit less policy change later in training.

3 EXPERIMENTS

We conduct experiments to demonstrate that: (1) LoGRA substantially improves training efficiency without compromising performance; (2) predicted-KL enables stable single-node RL training for more than 1,000 steps; and (3) ablation analyses provide deeper understanding of our method.

3.1 GENERAL EXPERIMENTAL SETUP

We primarily evaluate Qwen models ranging from 1.5B to 27B parameters, including Qwen2.5-Math-1.5B/7B and Qwen3.8-27B, on mathematical reasoning tasks. We also investigate long-horizon training of the 27B model on the Reasoning-Gym Hard mixture. Unless otherwise specified, we use REINFORCE with verifiable rewards, sampling one response per prompt and using a global running reward baseline for advantage estimation. This deliberately simple setup isolates the effects of our method from potential gains introduced by more sophisticated RL algorithms. By default, all experiments run on a single node equipped with eight H100 80GB GPUs and 128 CPU cores, with four GPUs allocated to FSDP training and the remaining four to independent generation engines.

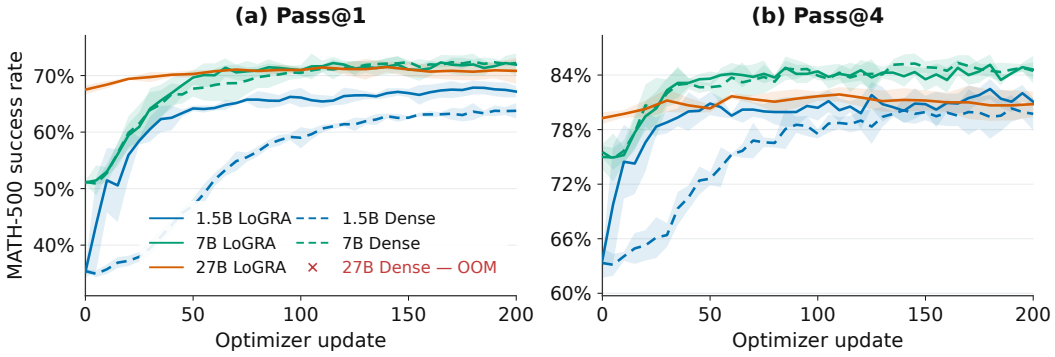


Figure 2: **Learning performance across model scales.** MATH-500 accuracy over the first 200 updates. Curves and bands show means and ± one sample standard deviation over three seeds.

3.2 MAIN EXPERIMENTS

We primarily compare LoGRA with dense-gradient training on DAPO-Math-7.5K. We evaluate on MATH-500 every five updates for the 1.5B and 7B models and every ten updates for the 27B model. Each batch contains 128 prompts, with one response per prompt, a 4,096-token context limit, and one policy update using a global reward baseline. LoGRA uses rank-256 RowAdam with mismatch subtraction and a predicted-KL budget annealed from 2×10^{-4} to 2×10^{-5}, whereas the dense baseline uses Adam with learning-rate annealing. We use RowAdam to enable a fairer comparison with dense Adam by retaining adaptive gradient scaling while keeping optimizer states compact (Appendix B). We run three runs for 200 updates at all three model sizes.

Table 1: **Accuracy, memory, and throughput of the main experiments.** Memory and throughput cover full runs; accuracy peaks use updates 0–200. Peak memory is the maximum of the three-seed mean curve.

Model	Method	n	Average memory ↓ (GiB/GPU)	Peak memory ↓ (GiB/GPU)	Updates/h ↑	Peak Pass@1 ↑ (%)	Peak Pass@4 ↑ (%)
1.5B	Dense	3	9.18 ± 0.00	9.19 ± 0.00	54.28 ± 0.40	63.77 ± 1.35	80.33 ± 1.47
1.5B	LoGRA	3	$\mathbf{7.18 \pm 0.10}$	$\mathbf{8.66 \pm 1.62}$	$\mathbf{55.44 \pm 0.41}$	$\mathbf{67.87 \pm 1.23}$	$\mathbf{82.47 \pm 1.10}$
7B	Dense	3	31.82 ± 0.00	31.85 ± 0.00	25.28 ± 0.68	72.48 ± 0.35	85.33 ± 0.83
7B	LoGRA	3	$\mathbf{17.29 \pm 0.34}$	$\mathbf{22.26 \pm 4.98}$	$\mathbf{25.73 \pm 0.28}$	72.33 ± 1.33	85.13 ± 1.14
27B	Dense	0			OOM		
27B	LoGRA	3	$\mathbf{51.54 \pm 0.17}$	$\mathbf{53.88 \pm 0.00}$	$\mathbf{6.99 \pm 0.02}$	$\mathbf{71.52 \pm 0.94}$	$\mathbf{81.87 \pm 0.81}$

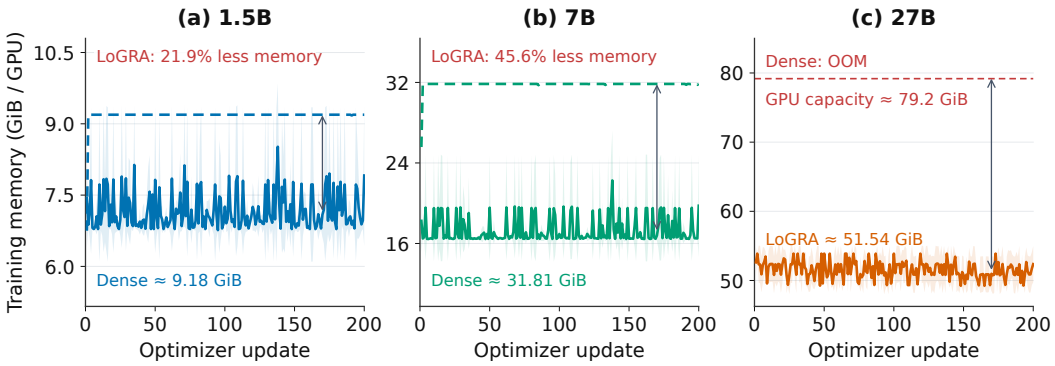


Figure 3: **Training memory across model scales.** Per-update peak allocated memory, averaged over three seeds, with ± one sample standard deviation. Arrows compare temporal means over updates 1–200 at 1.5B/7B; at 27B, the arrow shows headroom to device capacity.

Learning performance. At 1.5B, LoGRA achieves peak Pass@1 of 67.87%, compared with 63.77% for Dense; peak Pass@4 increases from 80.33% to 82.47%. At 7B, the methods attain similar peak accuracy: 72.33% versus 72.48% Pass@1 and 85.13% versus 85.33% Pass@4 for LoGRA and Dense, respectively. LoGRA also trains the 27B model successfully, reaching 71.52% Pass@1 and 81.87% Pass@4, whereas Dense exhausts GPU memory at its first Adam-state allocation. These results show that compressed updates preserve competitive learning performance in the evaluated configurations while making the largest model trainable under the same GPU budget.

Memory and throughput. Figure 3 shows that LoGRA lowers average per-update memory at 1.5B and 7B. Over the complete runs, average memory decreases from 9.18 to 7.18 GiB at 1.5B and from 31.82 to 17.29 GiB at 7B, reductions of 21.8% and 45.7%, respectively. Under the seed-averaged peak definition, LoGRA reduces peak memory from 9.19 to 8.66 GiB at 1.5B and from 31.85 to 22.26 GiB at 7B. At 27B, LoGRA uses 51.54 GiB on average and reaches a peak seed-averaged memory of 53.88 GiB; Dense fails before completing an update, precluding a measured memory-reduction ratio or throughput comparison. End-to-end update rates are similar where both methods complete training: 55.44 versus 54.28 updates/h at 1.5B and 25.73 versus 25.28 updates/h at 7B for LoGRA and Dense, respectively. The main systems benefit is reduced memory use and feasibility at 27B, with modest observed throughput gains at 1.5B and 7B.

3.3 Sustained 27B training on a single eight-GPU node

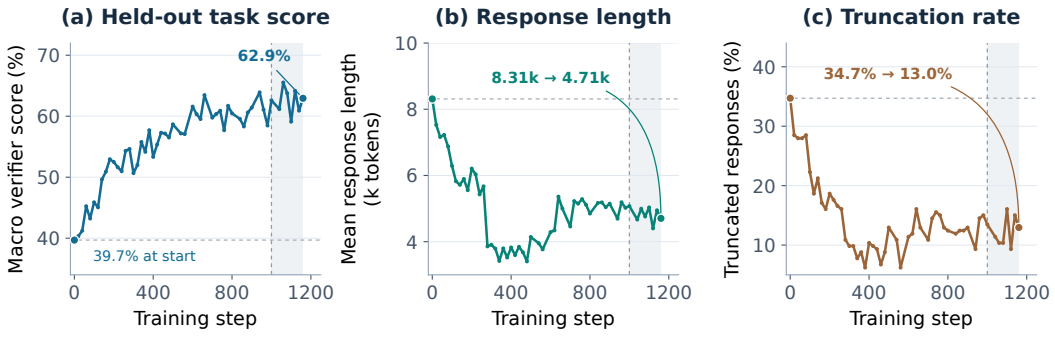


Figure 4: **Sustained 27B training on a single eight-GPU node.** Qwen3.8-27B trained with LoGRA on the Reasoning-Gym Hard mixture. **(a)** Macro-averaged verifier score across held-out tasks. **(b)** Mean evaluation response length. **(c)** Fraction of evaluation responses truncated at the 16,384-token context limit. All panels describe one training trajectory.

We demonstrate that LoGRA with predicted-KL enables stable 27B model training for over 1,000 steps on a single node. Specifically, we train Qwen3.8-27B on the Reasoning-Gym Hard mixture.

The dataset contains 38,948 training prompts and 193 hard held-out problems across eight tasks. We use rank-256 sketches and anneal the predicted-KL budget. Every 20 steps, we evaluate the model using the macro-averaged verifier score and monitor response length and truncation rate.

The held-out macro score increases from 39.69% initially to 62.94%, with a peak of 65.52% at step 1,060 (Figure 4a). The final five observed evaluations range from 59.11% to 64.16%, showing no sustained collapse. Over the same trajectory, mean response length decreases from 8,312 to 4,705 tokens, while the truncation rate falls from 34.72% to 12.95% (Figure 4b,c). This trajectory demonstrates that the LoGRA recipe can sustain learning of a 27B model beyond 1,100 logged training steps on one eight-GPU node, without sustained deterioration in the observed evaluation scores or runaway response-length growth.

3.4 Further Analysis

3.4.1 Design Choices in Low-Rank Gradient Compression

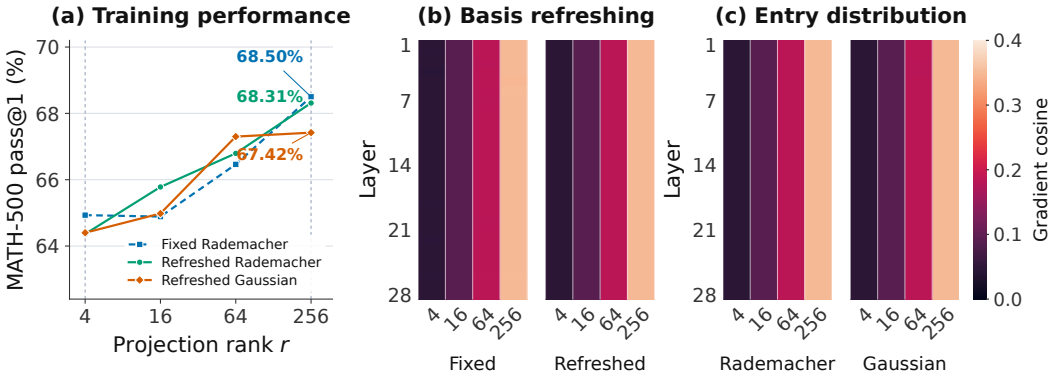


Figure 5: **Components of gradient compression.** **(a)** Training performance as a function of projection rank; **(b)** Gradient-reconstruction cosine for fixed and refreshed Rademacher bases. **(c)** The same diagnostic for refreshed Rademacher and Gaussian bases. Heatmap rows denote Transformer layers and columns denote ranks, with a shared $[0, 0.4]$ color scale; higher values indicate closer alignment with the dense gradient.

We next examine how projection rank, basis refreshing, and projection distribution affect gradient compression. All experiments use Qwen2.5-Math-1.5B and DAPO-Math-7.5k. We vary the projection rank over $r \in \{4, 16, 64, 256\}$ and compare fixed Rademacher, refreshed Rademacher, and refreshed Gaussian projections, where refreshing redraws the basis at every merge. Each plotted point reports one run, with MATH-500 pass@1 averaged over eight evaluations between steps 200 and 235. As shown in Figure 5(a), rank is the dominant factor: increasing the rank from 4 to 256 improves pass@1 by 3.02–3.95 percentage points across the three projection configurations. At rank 256, fixed Rademacher, refreshed Rademacher, and refreshed Gaussian projections achieve 68.50%, 68.31%, and 67.42%, respectively, although their relative ordering varies at lower ranks.

Figures 5(b) and 5(c) evaluate gradient reconstruction quality. We use the initialization and checkpoints at steps 160 and 280 from one fixed-basis rank-64 training trajectory. At each model state, we sample one response for each of 4,096 prompts and divide them into 32 minibatches of 128 responses. All projection conditions use the same responses and dense gradients. Fixed bases are reused across minibatches, whereas refreshed bases are redrawn for each minibatch. For each rank, we evaluate eight projection seeds and compute the Frobenius cosine similarity between the dense gradient G and its reconstruction for all 196 attention and MLP weight matrices. Each heatmap cell is averaged over the seven matrices in each Transformer layer, minibatches, model states, and projection seeds. Figure 5(b) compares fixed and refreshed Rademacher projections, whereas Figure 5(c) compares refreshed Rademacher and Gaussian projections. Reconstruction quality exhibits a strong dependence on rank: the mean cosine similarity increases from approximately 0.046 at rank 4 to 0.348 at rank 256. By contrast, the largest difference between corresponding layer–rank cells is only 0.0014 for fixed versus refreshed Rademacher projections and 0.0008 for refreshed Rademacher versus Gaussian

projections. Although the decrease in mean cosine under Gaussian projections is detectable at ranks 64 and 256, its magnitude remains below 2×10^{-4}. Overall, projection rank has a substantially greater effect on single-minibatch gradient reconstruction than either basis refreshing or projection distribution.

3.4.2 Comparison with LoRA

Table 2: **LoRA and LoGRA on Qwen2.5-Math-1.5B.** Values are mean ± sample standard deviation over three seeds.

Method	Average memory ↓ (GiB)	Peak memory ↓ (GiB)	Pass@1 ↑ (%)	Pass@4 ↑ (%)	Seconds/update ↓ (s)
LoRA	13.21 ± 0.07	13.38 ± 0.00	70.45 ± 3.09	81.40 ± 2.09	10.63 ± 0.10
LoGRA	$\mathbf{7.18 \pm 0.10}$	$\mathbf{8.66 \pm 1.62}$	68.47 ± 0.95	$\mathbf{81.80 \pm 1.06}$	11.31 ± 0.09

LoRA learns low-rank weight adapters, whereas LoGRA compresses gradients and updates to the model weights. We compare their memory use, accuracy, and update time. We compare rank-256 configurations of Qwen2.5-Math-1.5B trained on DAPO-Math-7.5k. For this comparison, we use the first 300 training steps of each method. LoRA uses AdamW with a constant learning rate of 2×10^{-4}; LoGRA uses a predicted-KL budget annealed from 2×10^{-4} to 2×10^{-5}.

Average memory is the temporal mean of the three-seed mean per-update memory curve; peak memory is its maximum. The per-step measurements are peak allocated training memory, excluding generation memory. Seconds/update measures the policy-training call, including call overhead but excluding response generation and weight synchronization.

Table 2 shows an obvious memory advantage for LoGRA under the recorded recipes, with average and peak memory of 7.18 and 8.66 GiB, compared with 13.21 and 13.38 GiB for LoRA. LoRA attains higher mean Pass@1 and a shorter policy-training call, while LoGRA attains a 0.40-percentage-point higher mean Pass@4. Thus, these results motivate gradient compression as an alternative when training memory is the limiting resource, but do not establish an accuracy or speed advantage.

3.4.3 KL Budget Control

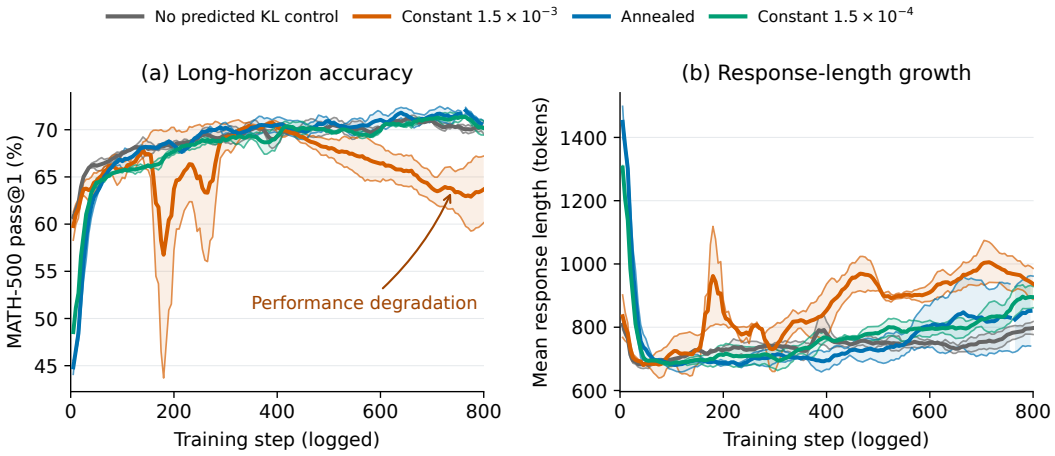


Figure 6: **Long-horizon accuracy and response length under different KL controls.** Thick solid lines show two-seed means, thin lines individual runs, and shading their range. Curves use five-evaluation moving means.

We then study whether predicted-KL step control improves long-horizon learning and how constant versus annealed KL budgets affect accuracy and response length. We use Qwen2.5-Math-1.5B on DAPO-Math-7.5k with rank-256 Rademacher projections refreshed every update, SGD, and a global

running reward baseline. We compare a fixed relative update norm of 6×10^{-5} without KL control, constant KL budgets of 1.5×10^{-3} and 1.5×10^{-4}, and a cosine-annealed budget from 2×10^{-4} to 2×10^{-5} over 1,500 steps. Each setting has two seeds; these KL-controlled settings do not subtract mismatch or use realised-KL feedback. Figure 6 plots MATH-500 pass@1 and response length over the first 800 logged steps, evaluating four responses per problem every five steps. Averaging raw evaluations over steps 700–800 within each run and then across seeds, pass@1 is 70.30% without KL control, 63.43% with the large constant budget, 71.03% with the small constant budget, and 71.47% with annealing. Corresponding response lengths are 774, 976, 858, and 832 tokens.

These results support the usefulness of predicted-KL step control when paired with a conservative budget: the small constant budget improves tail accuracy over no control while maintaining stable learning. A simple intuition is that limiting each policy change helps preserve previously learned behavior; an overly large budget weakens this protection and permits performance degradation. Both the small and annealed budgets remain stable over the observed horizon, with annealing achieving the highest mean tail accuracy. The no-control baseline also remains stable, and the separately collected annealed runs differ in their initial budget, so the results suggest a benefit from well-calibrated KL control without establishing that it is necessary for stability or that annealing alone explains the gain.

4 Related Works

Large language models have emerged as a foundation for general-purpose problem solving (Jimenez et al., 2024; Shao et al., 2024; Xie et al., 2024; Zhang et al., 2024; Yuan et al., 2026), with reinforcement learning (RL) becoming a critical post-training stage for advancing their capabilities (Guo et al., 2025; Yu et al., 2026; Hu et al., 2025). However, scaling RL is more challenging than scaling SFT, as RL repeatedly alternates among response generation, reward evaluation, and policy optimization, requiring careful coordination of computation and memory across these stages.

Recent work has addressed these challenges through asynchronous execution and more efficient resource management. For example, asynchronous RL overlap rollout with policy training, reducing GPU idle time between stages (Xu et al., 2026; Zhang et al., 2026a). Other works improve resource allocation and scheduling to accommodate the distinct computational demands of generation and training (Mei et al., 2025). Memory-saving techniques further enable larger models to fit within available hardware: sharding distributes training states across GPUs (Zhao et al., 2023), activation checkpointing reduces memory consumption by recomputing intermediate activations(Chen et al., 2016), and CPU offloading transfers selected training states to host memory (Rajbhandari et al., 2020). Our work complements them by compressing the gradients and optimization updates themselves, thereby reducing both training memory and communication overhead during policy synchronization.

Another line of work uses low-rank representations to reduce training costs. LoRA (Hu et al., 2021) freezes pretrained weights and trains low-rank adapters, reducing gradient and optimizer-state memory. Related approaches apply compression to gradients while keeping the original weights trainable. PowerSGD (Vogels et al., 2019) uses low-rank gradient approximations to reduce communication between distributed workers; GaLore (Zhao et al., 2024; Su et al., 2025) maintains optimizer statistics in a projected gradient space; and Flora (Hao et al., 2024) uses random projections to compress gradient accumulation and momentum. Building on these ideas, LoGRA targets LLM RL post-training by directly accumulating gradient sketches during backpropagation and combining factorized policy updates with predicted-KL step control. Its factorized update representation also supports synchronization between the trainer and rollout engine.

5 Conclusion

We presented LoGRA, a memory-efficient approach to LLM RL post-training that uses low-rank gradient sketches throughout gradient accumulation, weight updates, and policy synchronization. Our results show that compact gradient representations can substantially reduce training memory without sacrificing reasoning performance. In particular, LoGRA enables sustained training of a 27B model for over 1,100 steps on a single eight-GPU node, where dense Adam exceeds the available memory. Predicted-KL step control complements compression by regulating policy changes, with conservative budgets supporting stable long-horizon learning. Together, these findings highlight gradient representation as a practical lever for lowering the hardware barrier to RL post-training.

AI USE STATEMENT

Generative AI systems supported copyediting and clarity improvements during manuscript preparation. The research questions, method design, experiments, and interpretation of results were carried out by the authors. All AI-suggested revisions were manually reviewed, and the authors remain fully accountable for the paper's content.

ETHICS STATEMENT

Our experiments use publicly available models and benchmarks and involve neither human subjects nor sensitive personal information. By lowering the hardware requirements of RL post-training, LoGRA may broaden access to model development. As with other general-purpose training techniques, its downstream use requires appropriate safeguards and evaluation.

REPRODUCIBILITY STATEMENT

The main text specifies the LoGRA algorithm and experimental protocol. Appendix A provides the theoretical derivations, while Appendices C and D document dataset preparation, evaluation procedures, hyperparameters, and implementation choices. Main comparisons are repeated across multiple random seeds, with variability reported alongside average performance.

REFERENCES

Nemo rl: A scalable and efficient post-training library. https://github.com/NVIDIA-NeMo/RL, 2025. GitHub repository.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. *arXiv preprint arXiv:2402.03293*, 2024.

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. *arXiv preprint arXiv:2106.09685*, 2021.

Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Wenkai Fang, et al. Openrlhf: A ray-based easy-to-use, scalable and high-performance rlhf framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 656–666, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *Proceedings of Machine Learning and Systems*, 7, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pp. 1–16. IEEE, 2020.

John Schulman and Thinking Machines Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250929. https://thinkingmachines.ai/blog/lora/.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

DiJia Su, Andrew Gu, Jane Xu, Yuandong Tian, and Jiawei Zhao. Galore 2: Large-scale llm pre-training by gradient low-rank projection. *arXiv preprint arXiv:2504.20437*, 2025.

Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Binfeng Xu, Hao Zhang, Shaokun Zhang, Songyang Han, Mingjie Liu, Jian Hu, Shizhe Diao, Zhenghui Jin, Yunheng Zou, Michael Demoret, et al. Polar: Agentic rl on any harness at scale. *arXiv preprint arXiv:2605.24220*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2. 0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, et al. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents. *arXiv preprint arXiv:2603.18815*, 2026a.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. *arXiv preprint arXiv:2402.11359*, 2024.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 91437–91453, 2026b.

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

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

A Theoretical Derivations

A.1 From policy KL to changes in token scores

Equation 5 describes how KL changes with the update scale; Equation 6 gives its coefficient in terms of changes in token scores. We derive them in this order. Throughout, the proposed parameter update D and the nonempty set of contexts $\mathcal{P}$ are fixed. Each context s consists of a prompt and the response tokens preceding one next-token prediction. Both policies are evaluated on this same text.

Measure the change along an update. Let a be a scalar multiplier: $a = 0$ leaves the weights unchanged, while $a = 1$ applies the full proposal D. For vocabulary token v, write $p_v(a) = \pi_{\theta+aD}(v \mid s)$ and $p_v = p_v(0)$. We use natural logarithms and the same temperature $T > 0$ for both policies. Their KL at context s is

$$K_s(a) = \sum_v p_v \log \frac{p_v}{p_v(a)}. \tag{8}$$

A prime denotes a derivative with respect to the multiplier a. Thus $p_v'(0)$ measures how quickly token v's probability initially changes along D, whereas $K_s'(0)$ measures the initial slope of the KL. At $a = 0$ the two policies coincide, so $K_s(0) = 0$. Moreover,

$$K_s'(0) = -\sum_v p_v'(0) = 0, \tag{9}$$

because $\sum_v p_v(a) = 1$ for every a. Individual token probabilities can change to first order, but the KL has zero slope at the point where the distributions agree.

Obtain Equation 5: the quadratic dependence on step size. The second derivative $K_s''(0)$ measures the local curvature: how quickly the KL's slope changes as we move away from the current policy. Differentiating again gives

$$K_s''(0) = \sum_v \frac{[p_v'(0)]^2}{p_v} - \sum_v p_v''(0) = \sum_v p_v \left[\frac{\mathrm{d}}{\mathrm{d}a} \log p_v(a) \Big|_{a=0} \right]^2. \tag{10}$$

The term $\sum_v p_v''(0)$ vanishes because total probability remains one. For the local expansion, assume the logits have continuous, bounded third derivatives along D near $a = 0$. Taylor's formula then writes the KL for a small multiplier α as

$$K_s(\alpha) = K_s(0) + \alpha K_s'(0) + \frac{\alpha^2}{2} K_s''(0) + O(\alpha^3) = \frac{\alpha^2}{2} K_s''(0) + O(\alpha^3). \tag{11}$$

Here $O(\alpha^3)$ collects terms whose magnitude is bounded by a constant times $|\alpha|^3$ near zero, for fixed D. Averaging over the examined predictions gives

$$\frac{1}{|\mathcal{P}|} \sum_{s \in \mathcal{P}} K_s(\alpha) = \alpha^2 q(D) + O(\alpha^3), \qquad q(D) = \frac{1}{2|\mathcal{P}|} \sum_{s \in \mathcal{P}} K_s''(0). \tag{12}$$

Dropping the higher-order remainder gives Equation 5. Thus $q(D)$ is half the average KL curvature along the proposal: a larger value means that the same update scale changes the policy more. Within this approximation, halving the scale reduces KL by a factor of four.

Obtain Equation 6: express the curvature using token scores. Let $z_v(a; s)$ be the logit, or score, of token v at context s under weights $\theta + aD$. Define $\dot{z}_v(s) = \mathrm{d}z_v(a; s)/\mathrm{d}a|_{a=0}$, its initial rate of change along the proposal. Using

$$p_v(a) = \frac{\exp(z_v(a; s)/T)}{\sum_u \exp(z_u(a; s)/T)},$$

where u also ranges over vocabulary tokens, differentiating the log probability yields

$$\frac{\mathrm{d}}{\mathrm{d}a} \log p_v(a) \Big|_{a=0} = \frac{1}{T} \left(\dot{z}_v(s) - \sum_u p_u \dot{z}_u(s) \right). \tag{13}$$

The subtraction reflects that probabilities depend on relative scores: a token gains probability when its score increases faster than the probability-weighted average score. Substituting into Equation 10 gives

$$K_s''(0) = \frac{1}{T^2}\sum_v p_v\left(\dot{z}_v(s) - \sum_u p_u \dot{z}_u(s)\right)^2 = \frac{1}{T^2}\,\mathrm{Var}_{v\sim p(s)}[\dot{z}_v(s)],$$

where $p(s)$ collects the current token probabilities. This variance measures how differently the scores change, weighted by those probabilities. In particular, adding the same amount to every score leaves softmax probabilities unchanged; correspondingly, identical score-change rates give zero curvature. Substituting this expression into the definition of $q(D)$ above gives exactly Equation 6:

$$q(D) = \frac{1}{2|\mathcal{P}|T^2}\sum_{s\in\mathcal{P}} \mathrm{Var}_{v\sim p(s)}[\dot{z}_v(s)].$$

The average weights examined predictions equally. The probe evaluates the variance over the full vocabulary at each selected prediction; only the contexts are subsampled. It computes score-change rates before applying the update, using the factors $D_W = -\eta UA$ for each selected weight matrix. Neither a stored dense gradient nor a dense curvature matrix is needed.

A.2 Scaling an update to a KL budget

The multiplier α controls how much of the proposed update D we apply: $\alpha = 1$ keeps it unchanged, while $\alpha = 1/2$ halves it. Let $q(D)$ be the predicted KL for the full proposal and let $\delta > 0$ be the KL budget, the allowed average predicted change in the policy. We choose α by comparing these two quantities. If the proposal exceeds the budget, we shrink it; if it falls below the budget, we enlarge it.

The key is that predicted KL scales with the *square* of the update multiplier, as derived in Appendix A.1. To match the budget, we therefore solve $\alpha^2 q(D) = \delta$, giving $\alpha = \sqrt{\delta/q(D)}$ for finite positive $q(D)$. For example, if the proposal predicts four times the allowed KL, halving the update brings its predicted KL down to the budget. Conversely, if it predicts one quarter of the budget, doubling the update would reach the budget. We cap this enlargement at a chosen maximum multiplier $\alpha_{\max} > 0$, giving Equation 7:

$$\alpha = \min\left\{\alpha_{\max}, \sqrt{\delta/q(D)}\right\}. \tag{14}$$

The cap prevents a very small KL prediction from producing an excessively large update. This rule changes the update's size, not its direction.

B RowAdam

In the main experiments, we use RowAdam to make the comparison with dense Adam by retaining adaptive scaling based on past gradients. RowAdam is not identical to Adam: it adapts the scale of each row rather than each individual parameter, and does not maintain first-moment momentum.

Let S_t be the normalized gradient sketch at update t. For row i, RowAdam maintains a running estimate $v_{t,i}$ of its squared magnitude and uses it to scale the row:

$$\begin{aligned} v_{t,i} &= \beta_2 v_{t-1,i} + (1-\beta_2)\frac{1}{r}\sum_{j=1}^{r}(S_t)_{ij}^2, \\ (\widetilde{U}_t)_{i,:} &= \frac{(S_t)_{i,:}}{\sqrt{\widehat{v}_{t,i}} + \epsilon_t}, \qquad \widehat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2^t}. \end{aligned} \tag{15}$$

We initialize $v_{0,i} = 0$; the correction in $\widehat{v}_{t,i}$ accounts for this zero initialization. The parameter β_2 controls how much history is retained. Rows with consistently larger squared gradients receive smaller multipliers. After this row scaling, we rescale $\widetilde{U}_t$ to match the original sketch's Frobenius norm, yielding the factor U_t used for the weight update. Thus RowAdam adjusts the relative contribution of different rows while preserving the sketch's overall magnitude. We use $\beta_2 = 0.95$. The stabilizer ϵ_t is 10^{-8} times the mean of $\sqrt{\widehat{v}_{t,i}}$ across rows, with a positive numerical floor; zero sketches produce zero updates. The row statistics and their update counter persist across projection refreshes. Each selected matrix requires only d second-moment values, rather than one value per weight.

C Data Details

C.1 Mathematical reasoning

We construct DAPO-Math-7.5K from the DAPO-Math-17k training split, using the corresponding release on Hugging Face (Yu et al., 2026). The preparation script streams the source in its dataset order, retains the first occurrence of each prompt, and stops after collecting 7,500 unique prompts. The deduplication key is the content of the first prompt message.

Evaluation uses all 500 problems in the test split of MATH-500 (Lightman et al., 2023). Each problem is converted to a user message, and its reference answer is stored as a string for the verifier. Training and evaluation use the model's chat template. The training-subset deduplication described above operates within the training source; it is not a cross-dataset semantic decontamination procedure.

Evaluation generates four responses per problem at temperature 0.7 and top-p 0.7. Pass@1 is the mean binary correctness over these 2,000 responses. The verifier extracts and checks the final mathematical answer; an unparseable answer receives no correctness credit. Training generates one response per prompt at temperature 1.0 and top-p 1.0, with a 4,096-token total context limit.

C.2 Reasoning-Gym Hard mixture

Table 3: Reasoning-Gym Hard mixture.

Task	Training problems	Evaluation problems
`codeio`	4,868	25
`emoji_mystery`	4,752	22
`rectangle_count`	5,000	25
`word_ladder`	4,994	25
`pool_matrix`	5,000	25
`tsumego`	5,000	25
`sentence_reordering`	4,334	21
`manipulate_matrix`	5,000	25
Total	38,948	193

The long-horizon 27B experiment uses an eight-task Reasoning-Gym mixture with task-specific verifiers. We select tasks with mean verifier scores of approximately 0.25–0.60 in a preliminary evaluation of 89 tasks, excluding those whose failures mainly reflect response truncation or formatting errors. This provides a challenging training mixture with room for improvement.

For each task, we generate 5,000 candidate training problems and 25 evaluation problems at the default difficulty. We remove duplicate training questions and exclude training–evaluation overlaps; Table 3 reports the final counts. Each task's verifier returns a score in ([0,1]). We report the equally weighted mean of the eight task-average scores. Evaluation uses one response per problem, temperature 0.7, top-p 0.95, and a 16,384-token context limit.

D Hyperparameters

D.1 Main mathematical-reasoning experiments

Table 4 summarizes the settings for Table 1. Each run uses one node with eight H100 80GB GPUs and 128 CPU cores: four FSDP training GPUs and four independent generation engines (tensor-parallel degree one).

KL budget. For update $u = 1, \ldots, H$, the main experiments use

$$\delta_u = \delta_{\min} + \frac{\delta_{\max} - \delta_{\min}}{2} \left[1 + \cos\left(\pi \frac{u-1}{H-1} \right) \right], \tag{16}$$

with $H = 300$, $\delta_{\max} = 2 \times 10^{-4}$, and $\delta_{\min} = 2 \times 10^{-5}$. The 27B runs stop at 200 updates without shortening the schedule.

Table 4: Main-experiment settings. Schedule horizons count actual optimizer updates.

Setting	Dense	LoGRA
Training seeds	$1, 2, 42$	$1, 2, 42$
Prompts / responses per update	128 / 128	128 / 128
Microbatch per training GPU	1	1
Optimizer	Adam (zero weight decay)	RowAdam
Moment decay	$(\beta_1, \beta_2) = (0.9, 0.95)$	$\beta_2 = 0.95$; no first moment
Epsilon	10^{-8}, absolute	10^{-8}, relative
Projection	Not applicable	Rank 256, Rademacher
Projection refresh	Not applicable	Every update
Step-size rule	LR $10^{-6} \to 10^{-7}$	Predicted KL $2{\times}10^{-4} \to 2{\times}10^{-5}$
Schedule	9-update warmup; cosine decay	Cosine budget decay
Mismatch subtraction	Not applicable	Enabled; budget floor 50%
Evaluation interval	5 / 5 / OOM	5 / 5 / 10

Mismatch subtraction accounts for differences between rollout-engine and trainer probabilities at the same nominal weights. We average rollout-minus-trainer log probabilities over generated tokens in completed, nontruncated responses, clamp the estimate below at zero, and smooth it with moving-average weight 0.3. Denoting this estimate by κ_u, the effective budget is

$$\delta_u^{\text{eff}} = \max(\delta_u - \kappa_u,\, 0.5\delta_u). \tag{17}$$

Without a valid measurement, no subtraction is applied. This adjustment is a budgeting heuristic, not an exact decomposition of KL.

D.2 Long-horizon 27B training

The Reasoning-Gym run in Figure 4 uses seed 42, 128 prompts with one response each, microbatch size one, and a 16,384-token context limit. Training temperature and top-p are both 1.0. It uses rank-256 refreshed projections, SGD sketch updates, a global running baseline, and a KL budget annealed from 2×10^{-4} to 2×10^{-5} over 1,000 merges then held constant. Four training GPUs and four generation engines share one node. Evaluation is scheduled every 20 logged steps.

D.3 Reconstruction heatmaps.

We use initialization and steps 160 and 280 of one fixed-basis rank-64 trajectory. At each state, all conditions share 4,096 responses, grouped into 32 batches of 128, and their dense gradients. We measure reconstruction before optimizer transformations:

$$\widehat{G} = (GA^\top)A, \qquad \cos(G, \widehat{G}) = \frac{\sum_{i,j} G_{ij}\widehat{G}_{ij}}{\|G\|_F \|\widehat{G}\|_F}.$$

Each heatmap cell equally averages the seven attention/MLP matrices in a layer, 32 batches, three model states, and eight random projection realizations; 28 layers contain 196 matrices in total. Fixed bases are reused across batches; refreshed bases are redrawn.

D.4 LoRA

Table 2 uses rank 256 and Qwen2.5-Math-1.5B, with three runs per method. LoRA uses AdamW with constant learning rate 2×10^{-4}, moment decay $(0.9, 0.95)$, epsilon 10^{-8}, and zero weight decay. Both methods use 128 prompts with one response each, microbatch size one, and a 4,096-token context limit; LoGRA follows Table 4. LoRA statistics cover 300 logged steps, excluding records without an update from memory/time averages; LoGRA covers 300 actual updates. The different counters, optimizers, and schedules prevent isolating adapter parameterization alone.