PIGGYBACKING TRAINING ON INFERENCE SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training a model on the inference system that serves it would enable private on-device fine-tuning and turn idle inference fleet capacity into training capacity without separate training infrastructure. Inference hardware, however, often lacks the memory and interconnect needed for backpropagation. Zeroth-order optimization, such as evolution strategies (ES), uses only forward passes, but evaluating candidate-specific perturbations adds weight-update traffic or separate correction costs to batched inference. We address this problem with NOISYGEMM, a matrix-multiplication kernel that evaluates perturbed models without materializing their weight perturbations. Candidates use compact coefficient vectors over shared Hadamard bases, whose corrections are fused into the base GEMM. We integrate NOISYGEMM into vLLM's linear layers and run training through its existing scheduler, KV-cache manager, and sampler. On an A100, perturbed linear layers run at $1.1\times$–$1.3\times$ the latency of unperturbed cuBLAS GEMM. Using a separable CMA-ES variant, reinforcement learning of Qwen3 models from 1.7B to 30B-A3B improves on the base model on all six reasoning benchmarks and outperforms GRPO on 17 of 18 model–benchmark pairs with the same maximum completion length of 16k tokens. Compared to the state-of-the-art ES framework EGGROLL, our NOISYGEMM hosting CMA-ES achieves higher accuracy on AIME 2025 and lower latency when scaling the population size and adapter rank. Last but not least, we provide a demonstration of piggybacking CMA-ES training on a serving engine that replays the one-day BurstGPT live traffic.[1]

1 INTRODUCTION

Inference capacity for LLMs is growing rapidly, both in datacenter serving fleets and on consumer devices that run models locally. We ask whether a model can be trained *entirely* and *practically* on the inference system that serves it. On a local device, this would allow fine-tuning on private data without sending it to a training server. In the cloud, it would turn serving capacity that sits idle during low-demand periods into training capacity (Wang et al., 2025b; Stojkovic et al., 2025), without a separate training infrastructure.

Gradient-based training fits inference-oriented hardware poorly: inference hardware often lacks the memory, bandwidth, and interconnect that backpropagation needs for activations, gradients, and optimizer state (Malladi et al., 2023; Dettmers et al., 2023). Zeroth-order (ZO) optimization avoids these requirements: it estimates updates from forward passes of perturbed models, so no activations are kept for a backward pass, and because each perturbation is regenerated from a seed, workers exchange only seeds and scalar rewards (Spall, 1992; Salimans et al., 2017; Malladi et al., 2023). Recent work (Qiu et al., 2025; Korotyshova et al., 2025; Sarkar et al., 2025) shows that Zeroth-order Evolution Strategy (ES) can post-train LLMs (up to Qwen2.5-72B (Yang et al., 2024)) with results comparable to gradient-based methods such as GRPO (Shao et al., 2024) on grade-school math (e.g., GSM8k (Cobbe et al., 2021)) and puzzle-style arithmetic (e.g., Countdown).

Being forward-only, however, does not make ZO optimization efficient on inference systems. Modern LLM engines like vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) efficiently serve multiple requests by batching: one read of the weights serves every request in the batch. In ZO optimization like MeZO and Evolution Strategy, each request must be decoded by a different candidate model with its own perturbed weights, which breaks this sharing; we call this the *noise materialization*

[1]We will open-source our code upon the publication.

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.

problem. Existing ZO methods pay the extra latency cost for this problem in one of two ways (Table 1). MeZO and ESS forgo the batching, apply and score one candidate's perturbation at a time, each time re-reading the base weights (Malladi et al., 2023; Qiu et al., 2025) for materializing a new perturbation. ESSA and EGGROLL restore the shared read with low-rank perturbations but still form every candidate's weight in memory, where they compete with the KV cache for capacity and bandwidth (Korotyshova et al., 2025; Sarkar et al., 2025).

To address the *noise materialization* problem, we present NOISYGEMM, a matrix-multiplication kernel that evaluates candidate-specific weight perturbations without ever materializing them in GPU DRAM. Generating dense perturbations inside GEMM would avoid these memory writes but incur substantial random-number generation and arithmetic costs. We address both costs by representing each candidate perturbation with a compact coefficient vector over shared binary Hadamard bases. Binary entries reduce the cost of generating basis signs, while fast Hadamard transforms apply the bases using additions and subtractions and reuse transformed values across output blocks. The kernel fuses the resulting correction into the base GEMM's epilogue, while preserving shared weight reads across candidates and storing only their coefficients and a compact activation intermediate. As a result, NOISYGEMM re-enables fast, *batched* decoding for ZO training requests by reading their scalar seeds as input and generating their perturbation noise in-kernel.

Moreover, this coefficient representation also makes adaptive ZO optimization memory-efficient. The optimizer maintains its search center and per-coordinate statistics in coefficient space, so its state scales with the number of trainable coefficients rather than the number of model weights. Weighted sums of candidate noise can be accumulated by regenerating it from seeds, avoiding a separate stored population for the optimizer update. We instantiate this approach with a separable Covariance Matrix Adaptation Evolution Strategy (CMA-ES) variant (Ros & Hansen, 2008) for reinforcement learning with verifiable rewards. Our vLLM integration (Kwon et al., 2023) applies the perturbation corrections at the linear layers while reusing the engine's scheduler, KV-cache manager, sampler, and quantized base projections. Training then runs as an optimizer with a customizable reward function around ordinary generation calls.

Our contributions are:

- Identify noise materialization as the reason zeroth-order training cannot run at batched-inference efficiency: existing methods either give up shared weight reads or keep every candidate's perturbation in GPU memory (Table 1).
- Co-design the perturbation with the GEMM kernel so that no perturbation is ever materialized. Each candidate costs only R coefficients per adapted matrix instead of a weight copy or low-rank factors, and a perturbed forward pass runs within $1.1\times$–$1.3\times$ of unperturbed cuBLAS, while exactly preserving distances and angles in the coefficient space.
- Demonstrate NOISYGEMM also enables memory-efficient optimizers: because perturbations are regenerated from seeds, optimizer bookkeeping such as momentum is computed without storing any candidate's perturbation.
- Piggyback training on stock vLLM, where NOISYGEMM replaces only the linear layers; training reaches 72–73% of the engine's inference throughput.
- Report a training strategy built on separable CMA-ES that outperforms GRPO on 17 of 18 model–benchmark pairs across six reasoning benchmarks and converges above EGGROLL.

2 BACKGROUND AND RELATED WORK

2.1 RETHINKING INFERENCE SYSTEMS FOR TRAINING

The compute profile of LLM inference and training differs significantly, leading to the specialization of both software and hardware stacks (Patterson et al., 2021; Aminabadi et al., 2022). Backpropagation adds saved activations, gradients, and optimizer state to the model's memory footprint, and distributed training synchronizes gradient tensors across workers (Shoeybi et al., 2019; Rajbhandari et al., 2020; Zhao et al., 2023; Ryabinin et al., 2023). Inference avoids this training state and supports low-precision weights, allowing deployment on devices with less memory and on fleets without training-grade interconnects.

Table 1: **Cost of evaluating a population with zeroth-order methods,** per step for one $D \times D$ layer with population size of λ. NOISYGEMM avoids materializing the perturbed weights of size $r \times D$ for λ candidates and instead generates the noise from the Hadamard product between a coefficient vector ($O(\lambda r)$) and a shared base ($O(D)$).

Method	λ	Noise tensor in GPU DRAM	GPU DRAM read/write	Perturbation rank
MeZO (Malladi et al., 2023)	1 (pair)	✓	$O(\lambda D^2)$	full
ESS (Qiu et al., 2025)	30	✓	$O(\lambda D^2)$	full
ESSA (Korotyshova et al., 2025)	24–128	✓	$O(\lambda r D)$	r
EGGROLL (Sarkar et al., 2025)	256–8192	✓	$O(\lambda r D)$	r
NOISYGEMM (ours)	256–8192	✗	$O(D + \lambda r)$	r

Despite this gap, there is growing interest in repurposing idle inference capacity in datacenter fleets for training, since inference infrastructure is growing rapidly but is often underutilized because of diurnal usage patterns (Wang et al., 2025b; Stojkovic et al., 2025). Existing approaches reduce the training memory footprint with state offloading and parameter-efficient tuning (Rajbhandari et al., 2020; Zhao et al., 2023; Hu et al., 2022; Dettmers et al., 2023), or co-schedule training and serving on the same GPUs, sharing computation and device memory (Qiao et al., 2024; Miao et al., 2024a;b; Wang et al., 2025a). Both make conventional gradient-based training fit, either for small models by trading memory for compute, or on systems that can already train.

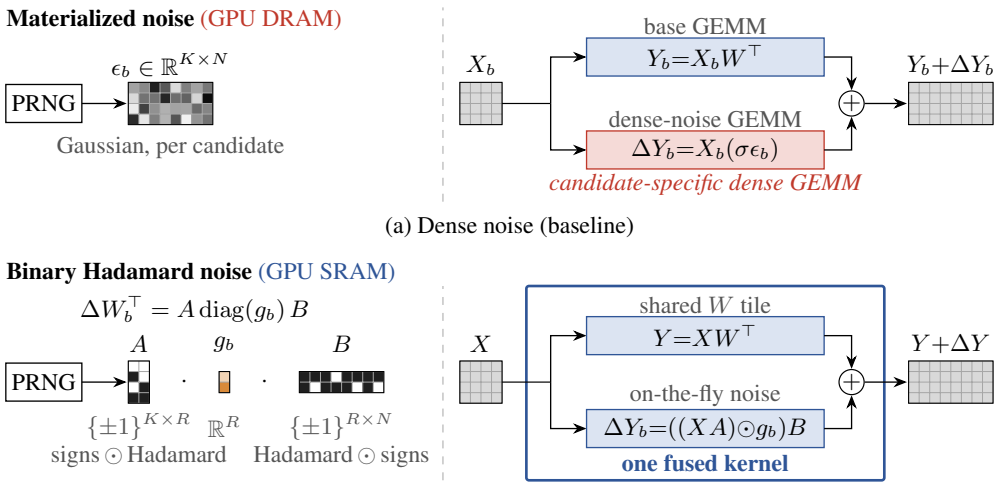
2.2 ZEROTH-ORDER OPTIMIZATION

Zeroth-order optimization (ZO) removes the backward-pass requirements by using only forward passes for training. ZO methods estimate update directions from forward evaluations of randomly perturbed parameters rather than from backpropagation (Spall, 1992; Nesterov & Spokoiny, 2017; Spall, 2003). Evolution strategies (ES) and natural-gradient variants operate on populations of such perturbations and aggregate compact scalar fitness values instead of gradient tensors (Salimans et al., 2017; Wierstra et al., 2014; Mania et al., 2018). With optimizer state replicated across workers, each candidate is identified by a seed and contributes a scalar reward, reducing optimization synchronization to $O(\lambda)$ scalars. ZO eliminates the backward pass and the activation storage required for backpropagation.

Recent work shows ZO methods can train LLMs at nontrivial scale. MeZO showed forward-only LLM fine-tuning is practical (Malladi et al., 2023); subsequent work improves variance, scaling, or second-order structure through layer-wise scaling, Hessian estimates, and subspace inference (Tan et al., 2025; Zhao et al., 2025; Feng & Huang, 2026). Beyond MeZO-style ZO fine-tuning, recent work applies evolutionary strategies to alignment, large-scale LLM fine-tuning, model merging, and parameter-efficient adapter search; related black-box prompt-tuning work optimizes prompts without model gradients (Korotyshova et al., 2025; Sarkar et al., 2025; Qiu et al., 2025; Akiba et al., 2025; Sun et al., 2024; Cheng et al., 2024). We use a separable CMA-ES variant (Ros & Hansen, 2008; Hansen, 2016) as our running example: unlike plain ES, it adapts a step size for every coordinate, and its separable form keeps this state linear in the number of parameters searched.

2.3 THE NOISE MATERIALIZATION PROBLEM

Batching amortizes one read of the weights over many requests, $Y_b = X_b W^\top$. In zeroth-order training, requests use candidate-specific weights, $Y_b = X_b(W + \Delta_b)^\top$, and this sharing breaks unless Δ_b is kept out of the weights. Existing implementations use in-place weight updates or separate corrections. MeZO, ESS, and ESSA regenerate a candidate's perturbation from its seed and apply it to the weights, so candidates run one at a time and each re-reads W (Malladi et al., 2023; Qiu et al., 2025; Korotyshova et al., 2025). EGGROLL restores the shared read with rank-r perturbations applied as a separate low-rank product, as in batched LoRA serving, but still forms every candidate's factors in memory (Sarkar et al., 2025). Table 1 compares the memory cost of evaluating a population of λ candidate perturbations of previous work against our NOISYGEMM.



(a) Dense noise (baseline)

(b) NoisyGEMM (this work)

Figure 1: **Overview of NoisyGEMM. (a)** Each candidate's dense Gaussian perturbation is written to DRAM and read by a separate correction GEMM, adding storage, memory traffic, and arithmetic to the shared base projection. **(b)** NoisyGEMM uses $\Delta W_b^\top = A\,\mathrm{diag}(g_b)\,B/\sqrt{K}$ with binary Hadamard bases A, B and R coefficients g_b; one fused kernel computes $XW^\top$ and the correction without forming ΔW_b. Each token row uses its candidate's coefficients; normalization by $1/\sqrt{K}$ is omitted in the diagram.

3 NoisyGEMM

The core idea of NoisyGEMM is to compute the effect of a candidate's weight perturbation inside the matrix multiplication kernel, without materializing the perturbation itself. For a layer with weights $W \in \mathbb{R}^{N\times K}$, an input row x_m within a batch of size M, and its candidate index $s(m)$, we compute

$$y_m = x_m W^\top + x_m \Delta W_{s(m)}^\top,$$

adding the second term while the base GEMM output is still being accumulated on chip instead of storing $W + \Delta W_b$ for every candidate in GPU memory (Figure 1).

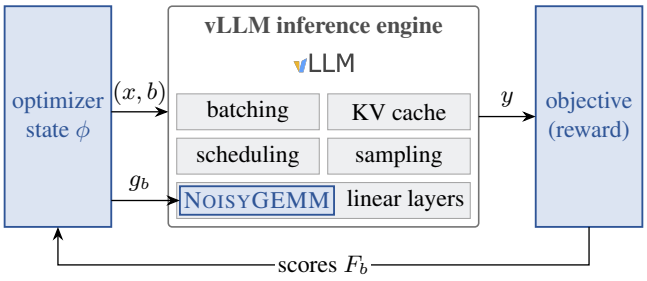
Generating dense noise inside the kernel is itself costly: a 16×16 by 16×16 matrix product performs 8,192 FLOPs, whereas Philox4x32-10 (Salmon et al., 2011) needs about 6.3K integer operations just to produce the random words for one 16×16 Gaussian noise tile (Appendix C). NoisyGEMM avoids this cost by building on two established tools, low-rank binary perturbations and randomized Hadamard transforms. Our contribution is a construction co-designed with the GEMM kernel: candidates differ only by per-row coefficients, the shared bases are applied by fast transforms whose output is reused across output blocks, and the correction is fused into the GEMM epilogue.

3.1 Low-rank binary perturbations

Following low-rank evolution strategies (Sarkar et al., 2025) and fixed-basis adapters (Balazy et al., 2025), we restrict each candidate's perturbation to a fixed R-dimensional linear subspace whose matrices have rank at most R:

$$\Delta W_b^\top = \frac{1}{\sqrt{K}} A\ \mathrm{diag}(g_b)\,B, \qquad \Delta y_b = \frac{\big((xA)\odot g_b\big)B}{\sqrt{K}}, \tag{1}$$

where $A \in \mathbb{R}^{K\times R}$ and $B \in \mathbb{R}^{R\times N}$ are shared by all candidates and only the coefficients $g_b \in \mathbb{R}^R$ vary. Candidates and learned updates are thus R scalars, and the perturbation is applied without being materialized in memory. As in SPSA (Spall, 1992) and binary random projections (Achlioptas, 2003), the entries of A and B are random signs, so each costs one random bit. Independent binary bases, however, still require two dense projections, 17.1% of the base GEMM's FLOPs at $K = 5120$, $N = 7168$, and $R = 512$.



(a) The training loop around vLLM

Require: engine $\mathcal{E}$, state ϕ
1: **for** $t = 1, \ldots, T$ **do**
2: $g_b \leftarrow \text{SAMPLE}(\phi, s_b),\ b \le \lambda$
3: Load $g_1, \ldots, g_\lambda$ into $\mathcal{E}$
4: $y_{p,b} \leftarrow \text{GENERATE}(\mathcal{E};\ x_p, b)$
5: $F_b \leftarrow \text{OBJECTIVE}(x_p, y_{p,b})$
6: $\phi \leftarrow \text{OPTIMIZE}\big(\phi, \{(s_b, F_b)\}\big)$
7: **end for**

(b) One training step

Figure 2: **Training by piggybacking an inference engine. (a)** The engine keeps its components (gray); training adds only the optimizer, the objective, and NOISYGEMM in the linear layers (blue). **(b)** A training step wraps batched generation with the optimizer and the objective.

3.2 FAST HADAMARD PROJECTIONS

We therefore build A and B from randomized Hadamard matrices, as in the fast Johnson–Lindenstrauss transform and Fastfood (Ailon & Chazelle, 2006; Le et al., 2013) and in recent LLM quantization kernels (Tseng et al., 2024; Ashkboos et al., 2024). A fast Walsh–Hadamard transform applies such a matrix with $q \log_2 q$ additions and subtractions and never stores it, so xA becomes a sign flip and one transform per input block. On the output side, every R-wide block of B uses the same H_R with different sign flips. The kernel therefore transforms the R scaled coordinates once per token and reuses the result for all N/R output blocks, changing only the signs.

The correction is then added in the base GEMM's epilogue, so a compact $M \times R$ intermediate replaces both the $K \times N$ candidate noise and the $M \times N$ output correction for a batch of M samples. This path adds about 0.10% of the base GEMM's FLOPs, compared with 17.1% for independent binary bases (measured latency in Section 5.5). For each adapted matrix, the coefficient-to-weight map preserves angles and satisfies $\|L(g) - L(h)\|_F = \sqrt{N}\,\|g - h\|_2$ (Appendix D). Isotropic noise in coefficient space therefore stays isotropic in weight space, so the optimizer's step size and per-coefficient scales directly set the size of the weight update.

4 ZEROTH-ORDER OPTIMIZATION WITH NOISYGEMM

NOISYGEMM lowers the cost of both population evaluation and optimizer updates without tying the coefficient interface to a particular optimizer. At each adapted layer, tokens in the engine's batch share the base GEMM across candidates, and NOISYGEMM adds their coefficient-specific corrections in the epilogue. The optimizer operates in L dimensions: the total coefficient count over all adapted matrices, and candidates can be described by seeds. This changes which optimizers are practical. Over dense weights, an optimizer that adapts a step size per coordinate must keep state for every model weight; over L coefficients (43,008 for Qwen3-1.7B) with fixed Hadamard base, the same optimizer state is negligible, so adaptive evolution strategies become affordable.

4.1 CMA-ES FOR RLVR

We instantiate this for reinforcement learning from verifiable rewards (RLVR) (Shao et al., 2024) with a CMA-ES variant (Hansen, 2016; Ros & Hansen, 2008) that adapts global and per-coefficient scales using $O(D)$ state. At step t, candidate b uses the coefficients

$$g_{t,b} = m_t + \sigma_t\, d_t \odot \varepsilon_{t,b}, \tag{2}$$

where m_t is the search center, σ_t the step size, d_t the per-coefficient scales, and $\varepsilon_{t,b}$ is regenerated from the candidate's seed. Candidates are drawn in antithetic pairs, $\pm\varepsilon_{t,b}$. The rollouts generated from a seed are then scored by a reward function. Outcome-based rewards are mostly binary and question difficulty varies widely within a dataset, so if each candidate answered its own random questions, its fitness would mostly reflect how easy those questions were. We instead tile a shared set of questions across candidates, so that each question is answered by many of them, and score every

rollout against the mean reward $\bar{r}_q$ of its question:

$$F_b = \frac{1}{|\mathcal{R}_b|} \sum_{(q,k)\in\mathcal{R}_b} \big(r_{q,b,k} - \bar{r}_q\big), \tag{3}$$

where $\mathcal{R}_b$ is the set of candidate b's rollouts. This is GRPO's group-relative advantage (Shao et al., 2024), except that the group spans perturbed policies rather than samples of one policy: questions that every candidate solves or fails thus contribute zero toward the aggregated update. The search center then takes a score-weighted step over the whole population with bias-corrected momentum, and σ_t, d_t adapt through evolution paths (Appendices F, G); Figure 2b summarizes one step.

The update itself needs weighted sums of the candidates' noise vectors, such as $\sum_b w_b\, \varepsilon_{t,b}$. The optimizer accumulates these sums by regenerating each noise vector from its seed, avoiding a separate $\lambda \times D$ noise buffer for the update.

4.2 Piggybacking Serving Systems for Training

An inference engine already provides almost everything a zeroth-order training step needs: it batches many requests, manages their KV caches, and samples completions at high throughput. NoisyGEMM adds candidate-specific forward passes through the linear layers while retaining the engine's scheduler, KV-cache manager, and sampler (Figure 2a). Its linear layers take one extra input, a table with one row of coefficients per candidate, and each request carries a tag that selects its row. Each candidate adds R coefficients per adapted matrix. The engine batches tagged requests through its existing scheduling and KV-cache mechanisms. The training loop is then an ordinary client of the engine (Figure 2b): it writes the candidates' coefficients, submits tagged prompts, scores the completions with an objective such as a verifier reward, and updates the optimizer. A request with a zero row sees the unperturbed model, so the same engine can keep serving regular traffic (§5.6).

We implement this on vLLM (Kwon et al., 2023) without modifying its scheduler, memory manager, or sampler: requests are tagged through vLLM's existing per-request adapter mechanism, and the correction is added to the output of the adapted projections, including quantized ones. We adapt the query, key, and value projections. Appendix E describes tensor-parallel sharding.

5 Evaluation

We evaluate whether training on an inference engine matches gradient-based post-training (Section 5.2 and Section 5.3), why it converges as it does (Section 5.4), where its efficiency comes from (Section 5.5), and whether it can share an engine with live traffic of inference requests (Section 5.6).

5.1 Setup

We train Qwen3-1.7B, Qwen3-4B, and Qwen3-30B-A3B (Yang et al., 2025) with reinforcement learning from verifiable rewards on math and coding data, and evaluate on AIME 2025 and 2026, HMMT February 2026, LiveCodeBench v5 and v6 (Jain et al., 2025), and HumanEval+ (Liu et al., 2023). NoisyGEMM runs inside vLLM (Kwon et al., 2023) with separable CMA-ES (§4), $R = 8$, and $\lambda = 512$ candidates, on A100 GPUs. The quality baseline is GRPO trained with verl (Sheng et al., 2025). All methods use the same prompts, verifier reward, and maximum completion length of 16k tokens in Table 2. The convergence and ablation curves use separate Qwen3-4B runs with 8192 rollouts per step and population sizes specified in the captions; they evaluate with a shorter 8k generation limit than Table 2, so the same base model starts at 35.8% on AIME 2025 rather than 53.5%. The system baselines use MeZO-style sequential weight perturbations and EGGROLL-style rank-16 factors (Malladi et al., 2023; Sarkar et al., 2025). Appendix H gives the datasets and experiment-specific protocols.

5.2 Training Quality

CMA-ES hosted by NoisyGEMM improves on the base model in all 18 model–benchmark pairs and outperforms GRPO in 17, with average gains of 5.7 and 1.9 percentage points, respectively, across all 18 pairs (Table 2). These gains are achieved by optimizing R coefficients per adapted

Table 2: **Accuracy (%) after training with the same 16k-token maximum completion length,** averaged over three runs. GRPO is trained with verl; NOISYGEMM runs a CMA-ES variant inside vLLM. Bold marks the highest mean accuracy for each model–benchmark pair.

Qwen3	Method	AIME25	AIME26	HMMT26	LCB v5	LCB v6	HumanEval+
1.7B	Base	32.5	32.5	21.8	55.5	52.5	76.8
	GRPO (verl)	35.2	35.5	24.2	59.9	55.9	82.0
	NOISYGEMM (ours)	**38.1**	**37.5**	**27.7**	**61.4**	**57.8**	**82.9**
4B	Base	53.5	54.6	31.4	68.2	66.5	78.5
	GRPO (verl)	56.3	58.8	33.7	76.5	72.6	**90.8**
	NOISYGEMM (ours)	**59.2**	**59.8**	**35.2**	**80.1**	**74.6**	89.6
30B-A3B	Base	59.6	66.9	34.5	79.1	74.9	88.4
	GRPO (verl)	62.6	69.3	39.8	79.8	74.6	89.2
	NOISYGEMM (ours)	**63.5**	**70.6**	**40.3**	**83.2**	**78.9**	**90.2**

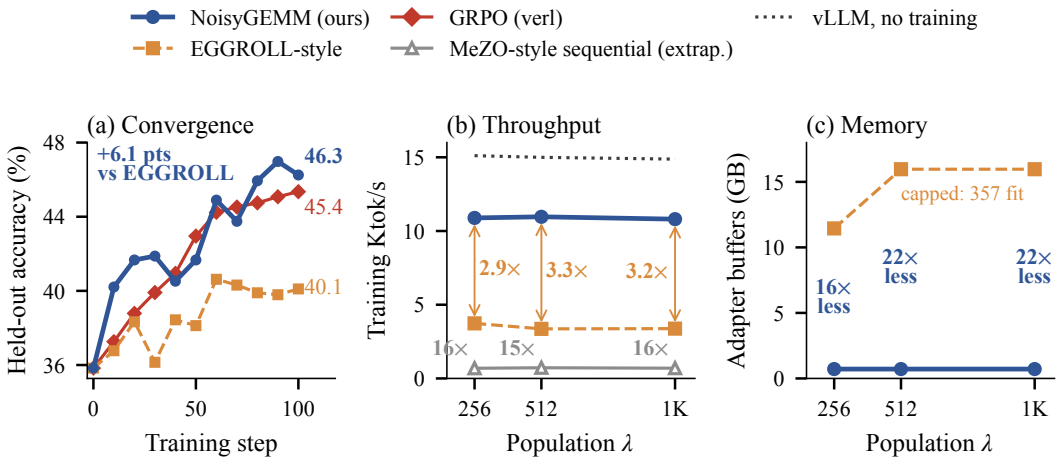


Figure 3: **Convergence, training throughput, and memory overhead. (a)** Qwen3-4B, AIME 2025, 8192 rollouts per step; the adapted run uses $\lambda = 512$ and 16 rollouts per candidate, and EGGROLL uses rank-16 factors, as in (b, c) and Figure 5. **(b, c)** Qwen3-1.7B on one A100, including perturbation setup and update; dotted: vLLM without training; the sequential baseline is extrapolated from 32 candidates. Bold: NOISYGEMM's advantage.

matrix: just 43,008 trainable coefficients for Qwen3-1.7B. The base weights remain frozen throughout training. Fine-tuning has a low intrinsic dimension (Li et al., 2018; Aghajanyan et al., 2021), and fixed-basis adapters with few trainable parameters match much larger ones (Balazy et al., 2025). In a space this small, the 2048 candidates of each step probe about 5% of the directions for Qwen3-1.7B, which keeps the variance of the zeroth-order estimate low. On Qwen3-30B-A3B, CMA-ES hosted by NOISYGEMM improves LiveCodeBench v5 and v6 by 4.1 and 4.0 percentage points over the base model, compared with $+0.7$ and -0.3 for GRPO. To the best of our knowledge, this is the first work that shows a zeroth-order RL optimization can improve the strong Qwen3 series of models and outperform GRPO on competition-level math and coding benchmarks.

5.3 ABLATION STUDIES

Step-size adaptation improves final accuracy over both fixed-step settings (Figure 4b). A large fixed step ($\sigma = 0.005$) learns fastest early, 1.3 points ahead of the adapted run over steps 10–30, but averages 41.2% over steps 90–120; the smaller step ($\sigma = 0.001$) averages 38.4%. A fixed step must compromise between the early phase, which rewards large moves, and the late phase, where large moves overshoot. Adapting the step sizes σ_t and d_t shrinks steps as training converges, and the adapted run averages 3.9 percentage points above the better fixed-step run over steps 90–120.

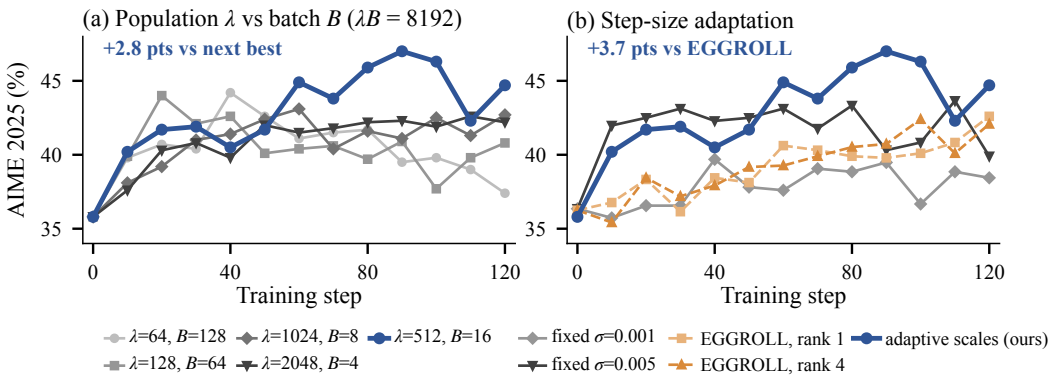


Figure 4: **Ablations** (Qwen3-4B, AIME 2025). **(a)** Population λ against rollouts per candidate B at $\lambda B = 8192$. **(b)** Adaptive scales versus fixed isotropic scales ($d = \mathbf{1}$, $\sigma \in \{0.001, 0.005\}$) and EGGROLL ranks 1 and 4. Gains are means over steps 90–120.

A population of 512 gives the highest final accuracy (Figure 4a). At 8192 rollouts per step, increasing λ trades more search directions for fewer reward observations per candidate. Populations of 64–128 score each candidate precisely but sample few of the tens of thousands of directions; they peak early and decline. Populations of 1024–2048 receive only 4–8 rewards per candidate, so their ranking is noisy and they reach lower final accuracy. A population of 512, with 16 rollouts per candidate, achieves the highest mean accuracy over steps 90–120. This optimum already exceeds the 357 rank-16 adapters that fit in 16 GB (Appendix H), so materialized perturbations must be evaluated in rounds; the throughput comparison in §5.5 includes this cost.

5.4 Convergence

Per step (Figure 3a), NoisyGEMM leads GRPO early (41.7% against 38.8% after 20 steps) and at the end (46.3% against 45.4%), while EGGROLL, with the same rollouts, stalls near 40%. The early lead is consistent with the small search space of §5.2. The gap to EGGROLL reflects the optimizer that each representation permits. Because all NoisyGEMM candidates share the same bases, a coefficient means the same weight direction for every candidate, so one scale per coefficient, R values per adapted matrix, serves the whole population and separable CMA-ES can adapt it. EGGROLL draws fresh low-rank factors for each candidate, which leaves no shared coordinates to scale, so it uses a single global step size. Section 5.3 compares adaptive and fixed step sizes.

5.5 Where the Efficiency Comes From

NoisyGEMM's throughput is flat in the population: on Qwen3-1.7B with Countdown rollouts, it sustains 10.8–11.0K tokens/s from 256 to 1024 candidates (Figure 3b), because each candidate adds only R coefficients per adapted matrix to the batch. The baselines show the two costs of noise materialization from §2.3. The MeZO-style baseline applies each perturbation to the weights, so it decodes only one candidate's four sequences at a time; this small batch, not the perturbation itself, makes it 15–16× slower. The EGGROLL-style baseline batches candidates, but its rank-16 adapters occupy 11.5–16 GB (Figure 3c) and shrink the KV cache by 20–27%, so fewer sequences fit in a batch and it runs 2.9–3.3× slower.

The same pattern holds inside a single linear layer (Figure 5). NoisyGEMM's correction is a per-token Hadamard transform whose cost is independent of the number of candidates and grows only as $R \log R$ with the number of coefficients: it stays at 1.29×–1.31× cuBLAS latency from $\lambda = 256$ to 2048 and at 1.15×–1.21× from $R = 128$ to 1024. Materialized perturbations are instead read once per candidate, so their cost grows with both: full-rank perturbations cost 15–116× cuBLAS, and low-rank factors up to 3.5× at rank 16 and 10× at rank 512 (Appendix I). End to end, the prototype reaches 72–73% of unperturbed vLLM throughput; the remaining gap comes from tagging candidates through vLLM's LoRA slots, not from the kernel (Appendix H).

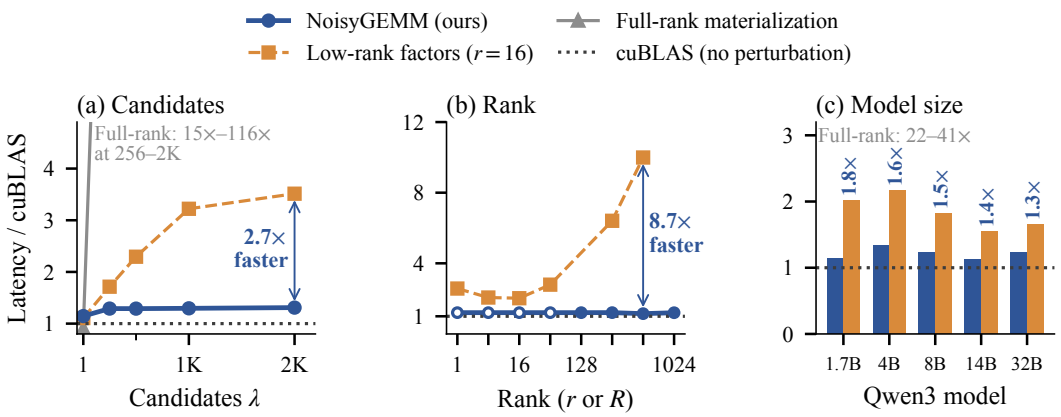


Figure 5: **Cost of one perturbed linear layer** (A100, relative to cuBLAS). The full-rank baseline stores perturbed weight copies; the EGGROLL-style baseline stores low-rank factors. **(a)** Qwen3-1.7B query/key/value, 2048 tokens over λ candidates. **(b)** Rank at $\lambda = 256$; below 128, NoisyGEMM runs $R = 128$ with the extra coefficients zero (hollow). **(c)** Qwen3 models at $\lambda = 256$; numbers above bars give the low-rank baseline's latency divided by NoisyGEMM's.

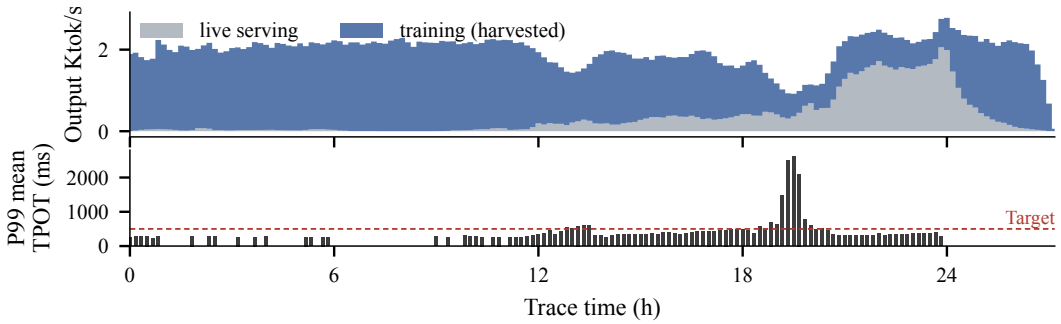


Figure 6: **Training alongside live traffic** (BurstGPT, replayed 144× faster with 2.5% of requests, 10-minute buckets). Top: live and training output-token throughput. Bottom: P99 of per-request mean TPOT, grouped by arrival time; dashed: 500 ms target.

5.6 Piggybacking Live Serving

A training request differs from a live one only in its coefficient row, so training fills the same batches instead of a separate replica, and admitting it becomes a scheduling decision. Replaying one day of BurstGPT traffic (Wang et al., 2025b) on Llama-3.1-8B with a target P99 TPOT of 500 ms (Figure 6), the shared engine generated 1.1M training output tokens, 82% of all output tokens, mostly during the overnight trough when live traffic leaves the engine underused. The cost appears at the two traffic peaks, where P99 TPOT reached 2.6 s. Without training, P99 TPOT at the same peaks also exceeds the target, so these violations stem mainly from live load rather than from training (metrics in Appendix H).

6 Conclusion

NoisyGEMM enables post-training on a serving engine by computing candidate-specific corrections from shared Hadamard bases and compact coefficients. Perturbed linear layers run at 1.1×–1.3× cuBLAS GEMM latency on an A100. On Qwen3 models from 1.7B to 30B-A3B, training outperforms GRPO on 17 of 18 model–benchmark pairs with the same maximum completion length of 16k tokens, and training requests share an engine with live traffic.

Limitations. Updates lie in a fixed R-dimensional subspace per matrix, and we leave sweeps over R and the basis seed to future work. Our efficiency results measure per-step cost, not time-to-accuracy for Table 2. Our results use verifiable rewards on math and coding and do not cover preference-based or otherwise non-verifiable objectives.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, including the prototype system and the benchmark and plotting code, to provide feedback on the research methodology and experimental design, to assist in writing the proof in Appendix D, and to help interpret results. We did not use generative AI tools to generate data or to formulate the method or its mathematical claims, and translation, dataset cleaning, and qualitative data analysis are not applicable. We also used generative AI tools to draft and edit text, create and edit figures, search and summarize literature, and format and verify references. All AI-assisted code, text, figures, and references were reviewed by the authors, who take responsibility for the final content.

ETHICS STATEMENT

Our evaluation uses publicly released models, datasets, and workload traces, and contains no personal data. Concurrent training and serving is evaluated by replaying a public request trace; because training can increase serving latency, we report request latency against the SLO throughout the replay.

REPRODUCIBILITY STATEMENT

Section 3.1 and Appendix D specify the Hadamard weight perturbations and linear-layer output computation. Section 4 describes the optimizer instantiation, and Appendix G specifies its updates and the weighted moment reductions. Appendix H describes the datasets and experiment-specific protocols, and Appendices I and E describe the kernel benchmarks and parallel execution. We will release the kernel, the vLLM integration, run configurations, and raw logs.

REFERENCES

Dimitris Achlioptas. Database-friendly random projections: Johnson–Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003. doi: 10.1016/S0022-0000(03)00025-4.

Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pp. 7319–7328, 2021.

Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast Johnson–Lindenstrauss transform. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pp. 557–563, 2006. URL https://www.cs.princeton.edu/~chazelle/pubs/stoc06.pdf.

Takuya Akiba, Makoto Shing, Yujin Tang, Qi Sun, and David Ha. Evolutionary optimization of model merging recipes. *Nat. Mac. Intell.*, 7(2):195–204, 2025.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 46:1–46:15. IEEE, 2022. doi: 10.1109/SC41404.2022.00051. URL https://dblp.org/rec/conf/sc/AminabadiRALLZRSZRH22.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. QuaRot: Outlier-free 4-bit inference in rotated LLMs. In *Advances in Neural Information Processing Systems*, 2024.

Klaudia Balazy, Mohammadreza Banaei, Karl Aberer, and Jacek Tabor. Lora-xs: Low-rank adaptation with extremely small number of parameters. In *ECAI*, Frontiers in Artificial Intelligence and Applications, pp. 3194–3201. IOS Press, 2025.

Jiale Cheng, Xiao Liu, Kehan Zheng, Pei Ke, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. Black-box prompt optimization: Aligning large language models without model training. In *ACL (1)*, pp. 3201–3219. Association for Computational Linguistics, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

D. E. Shaw Research. Random123: Philox implementation. https://github.com/DEShawResearch/random123/blob/main/include/Random123/philox.h. Accessed September 25, 2026.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*, 2023. URL https://dblp.org/rec/conf/nips/DettmersPHZ23.

Jian Feng and Zhihong Huang. Robust and efficient zeroth-order LLM fine-tuning via adaptive bayesian subspace optimizer. *CoRR*, abs/2601.01452, 2026.

Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016. URL https://dblp.org/rec/journals/corr/Hansen16a.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://dblp.org/rec/conf/iclr/HuSWALWWC22.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025. URL https://dblp.org/rec/conf/iclr/JainHGLYZWSSS25.

Daria Korotyshova, Boris Shaposhnikov, Alexey Malakhov, Nikita Surnachev, George Bredis, Alexey Gorbatovski, Viacheslav Sinii, and Daniil Gavrilov. ESSA: evolutionary strategies for scalable alignment. *CoRR*, abs/2507.04453, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

Quoc V. Le, Tamás Sarlós, and Alexander J. Smola. Fastfood – computing Hilbert space expansions in loglinear time. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 244–252, 2013. URL https://proceedings.mlr.press/v28/le13.html.

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.

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. URL https://dblp.org/rec/conf/nips/LiuXW023.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. In *Advances in Neural Information Processing Systems*, 2023. URL https://dblp.org/rec/conf/nips/MalladiGNDL0A23.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 1805–1814, 2018. URL https://dblp.org/rec/conf/nips/ManiaGR18.

Xupeng Miao, Gabriele Oliaro, Xinhao Cheng, Mengdi Wu, Colin Unger, and Zhihao Jia. Flexllm: A system for co-serving large language model inference and parameter-efficient finetuning. *CoRR*, abs/2402.18789, 2024a. doi: 10.48550/arXiv.2402.18789. URL https://dblp.org/rec/journals/corr/abs-2402-18789.

Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1112–1127. ACM, 2024b. doi: 10.1145/3620665.3640411. URL https://dblp.org/rec/conf/asplos/MiaoSDXL0J24.

Yurii E. Nesterov and Vladimir G. Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017. doi: 10.1007/s10208-015-9296-2. URL https://dblp.org/rec/journals/focm/NesterovS17.

David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021. URL https://dblp.org/rec/journals/corr/abs-2104-10350.

Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. Conserve: Harvesting GPUs for low-latency and high-throughput large language model serving. *CoRR*, abs/2410.01228, 2024. doi: 10.48550/arXiv.2410.01228. URL https://dblp.org/rec/journals/corr/abs-2410-01228.

Xin Qiu, Yulu Gan, Conor F Hayes, Qiyao Liang, Yinggan Xu, Roberto Dailey, Elliot Meyerson, Babak Hodjat, and Risto Miikkulainen. Evolution strategies at scale: Llm fine-tuning beyond reinforcement learning. *arXiv preprint arXiv:2509.24372*, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 20:1–20:16. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00024. URL https://dblp.org/rec/conf/sc/RajbhandariRRH20.

Raymond Ros and Nikolaus Hansen. A simple modification in CMA-ES achieving linear time and space complexity. In *Parallel Problem Solving from Nature – PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pp. 296–305. Springer, 2008. doi: 10.1007/978-3-540-87700-4_30. URL https://dblp.org/rec/conf/ppsn/RosH08.

Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. SWARM parallelism: Training large models can be surprisingly communication-efficient. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 29416–29440. PMLR, 2023. URL https://dblp.org/rec/conf/icml/RyabininDDB23.

Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017. URL https://dblp.org/rec/journals/corr/SalimansHCS17.

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 16:1–16:12. ACM, 2011. doi: 10.1145/2063384.2063405. URL https://dblp.org/rec/conf/sc/SalmonMDS11.

Bidipta Sarkar, Mattie Fellows, Juan Agustin Duque, Alistair Letcher, Antonio León Villares, Anya Sims, Dylan Cope, Jarek Liesen, Lukas Seier, Theo Wolf, Uljad Berdica, Alexander David Goldie, Aaron C. Courville, Karin Sevegnani, Shimon Whiteson, and Jakob Nicolaus Foerster. Evolution strategies at the hyperscale. *CoRR*, abs/2511.16652, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/arXiv.2402.03300. URL https://dblp.org/rec/journals/corr/abs-2402-03300.

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. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075. URL https://dblp.org/rec/conf/eurosys/ShengZYWZZPL025.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL https://dblp.org/rec/journals/corr/abs-1909-08053.

James C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, 1992. doi: 10.1109/9.119632.

James C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley Series in Discrete Mathematics and Optimization. Wiley-Interscience, 2003. ISBN 978-0-471-33052-3.

Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing LLM inference clusters for performance and energy efficiency. In *Proceedings of the 31st IEEE International Symposium on High-Performance Computer Architecture*, pp. 1348–1362. IEEE, 2025. doi: 10.1109/HPCA61900.2025.00102. URL https://dblp.org/rec/conf/hpca/StojkovicZGTC25.

Jingwei Sun, Ziyue Xu, Hongxu Yin, Dong Yang, Daguang Xu, Yudong Liu, Zhixu Du, Yiran Chen, and Holger R. Roth. Fedbpt: Efficient federated black-box prompt tuning for large language models. In *ICML*, Proceedings of Machine Learning Research, pp. 47159–47173. PMLR / OpenReview.net, 2024.

Qitao Tan, Jun Liu, Zheng Zhan, Caiwei Ding, Yanzhi Wang, Jin Lu, and Geng Yuan. Harmony in divergence: Towards fast, accurate, and memory-efficient zeroth-order LLM fine-tuning. *CoRR*, abs/2502.03304, 2025.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with Hadamard incoherence and lattice codebooks. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. Colocating ML inference and training with fast GPU memory handover. In *Proceedings of the 2025 USENIX Annual Technical Conference*, pp. 1657–1675. USENIX Association, 2025a. URL https://dblp.org/rec/conf/usenix/WangWH025.

Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize LLM serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5831–5841. ACM, 2025b. doi: 10.1145/3711896.3737413. URL https://dblp.org/rec/conf/kdd/0003CLKFZZT0GW025.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(1):949–980, 2014. doi: 10.5555/2627435.2638566. URL https://dblp.org/rec/journals/jmlr/WierstraSGSPS14.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin

Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *CoRR*, abs/2412.15115, 2024. doi: 10.48550/arXiv.2412.15115. URL https://dblp.org/rec/journals/corr/abs-2412-15115.

An Yang et al. Qwen3 technical report. *CoRR*, abs/2505.09388, 2025. doi: 10.48550/arXiv.2505.09388. URL https://dblp.org/rec/journals/corr/abs-2505-09388.

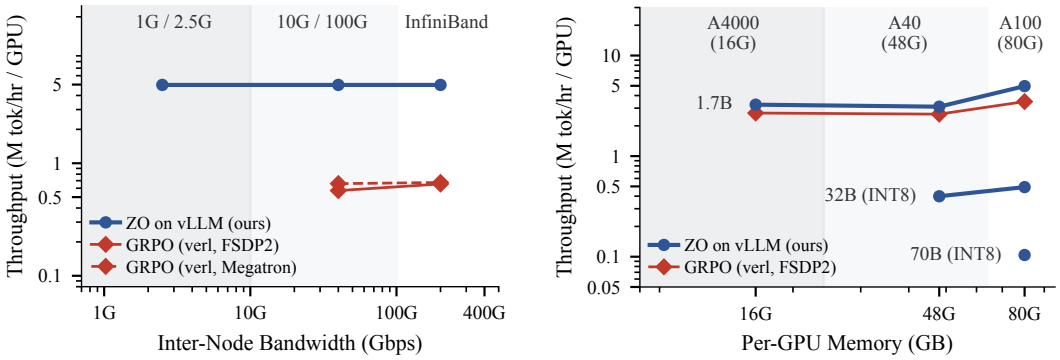
Yanjun Zhao, Sizhe Dang, Haishan Ye, Guang Dai, Yi Qian, and Ivor W. Tsang. Second-order fine-tuning without pain for llms: A hessian informed zeroth-order optimizer. In *ICLR*. OpenReview.net, 2025.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023. doi: 10.14778/3611540.3611569. URL https://dblp.org/rec/journals/pvldb/ZhaoGVLHXWSOSDB23.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024. URL https://dblp.org/rec/conf/nips/ZhengYXS0YCKSGB24.

A Why Gradient-Based Training Fits Inference Hardware Poorly

Figure 7 shows two constraints of inference hardware. **Interconnect.** Gradient-based training synchronizes gradient-sized tensors every step, so its throughput depends on the network; zeroth-order training exchanges only seeds, rewards, and a small coefficient state (Appendix E). **Memory.** Backpropagation stores activations, gradients, and optimizer state besides the weights, which rules out larger models on small-memory devices; forward-only training needs only the weights, the KV cache, and the coefficient state.



(a) Interconnect: training throughput vs. inter-node bandwidth.

(b) Memory: per-GPU throughput vs. device memory.

Figure 7: **Interconnect and memory constraints of inference hardware. (a)** First-order baselines (verl with FSDP2 or Megatron) synchronize gradients every step and lose throughput at Ethernet bandwidths, while zeroth-order training on vLLM exchanges no gradient tensors. The curves are an analytical model of Qwen3-8B training on 8 GPUs: the first-order curves add the gradient-synchronization time at each bandwidth to a reference step time, and the zeroth-order line is a measured throughput. **(b)** Peak per-GPU throughput on A4000 (16 GB), A40 (48 GB), and A100 (80 GB) GPUs. First-order training of Qwen3-1.7B runs on all three, but 32B and 70B models (INT8) fit only with forward-only training.

B MULTI-NODE SCALING

Because nodes exchange no gradients, adding nodes adds rollout capacity (Figure 8a). Training Qwen3-8B with one A100 per node over Ethernet, the time per step falls from 600 s on one node to 38.4 s on 24 nodes (15.6×) and 23.4 s on 32 nodes (25.7×), while GRPO in verl takes 448, 229, and 639 s on 4, 8, and 16 nodes. On a single A100 (Figure 8b, c), decoding Llama-3.1-8B with every request perturbed reaches 1,689 tokens/s at batch size 320, 84% of unperturbed inference, at unchanged peak memory. At batch size 64, it reaches 768 tokens/s versus 235 for the MeZO-style baseline, a 3.3× speedup.

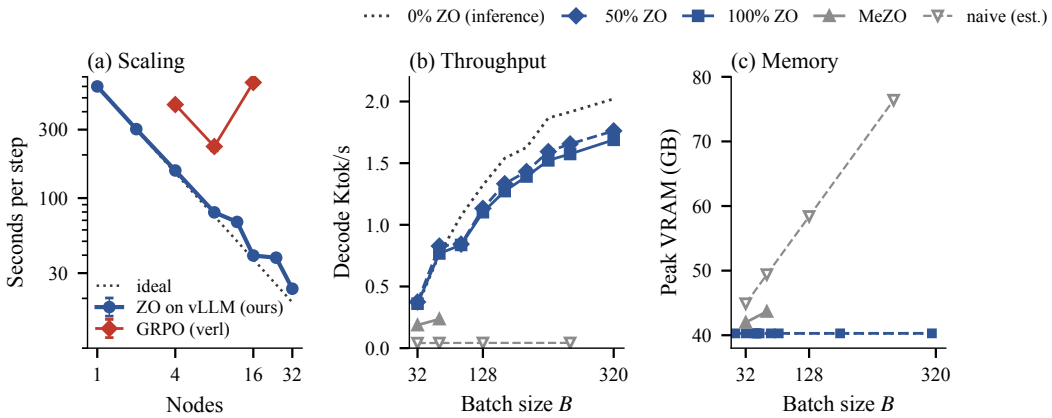


Figure 8: **Multi-node scaling and serving throughput. (a)** Seconds per training step versus nodes (Qwen3-8B, one A100 per node, Ethernet); dotted: linear scaling. **(b)** Decode throughput versus batch size with 0%, 50%, and 100% of requests perturbed, and for MeZO (Llama-3.1-8B, A100). **(c)** Peak allocated memory. "Naive" estimates one materialized weight copy per candidate.

C NOISYGEMM CONSTRUCTION AND COST

PRNG cost. Philox4x32-10 needs 98 integer operations per evaluation of four 32-bit words (Salmon et al., 2011; D. E. Shaw Research), so the 256 Gaussian entries of one 16×16 noise tile cost 6,272 operations, against 8,192 FLOPs for a 16×16 matrix product. Binary entries need one bit each, so 256 entries take two evaluations instead of 64, a 32× reduction, with no Gaussian conversion.

Input projection. We partition the input into $T = K/k_b$ blocks of power-of-two size $k_b \geq R$. Every block uses the same R Hadamard columns with block-specific sign flips (Eq. (5)), so the kernel sign-flips each block, applies a fast Hadamard transform, keeps the R selected coordinates, sums them over blocks, and scales the result by the candidate's coefficients, $h = (xA) \odot g_b$.

Output projection. Every R-wide output block of B is H_R with different sign flips, so the kernel computes $u = hH_R/\sqrt{K}$ once per token and expands it as $\Delta y_b = [\, u \odot d_2^{(1)} \mid \cdots \mid u \odot d_2^{(N/R)} \,]$. Randomness enters only through the signs and the selected columns, so candidates differ only in g_b.

Kernel. A persistent-grid kernel runs the transforms and the base GEMM in one launch (Algorithm 1). The transforms use registers and shared memory and pass an $M \times R$ intermediate through global memory; the GEMM epilogue reads it, applies the output signs, and adds the correction before storing the output.

Arithmetic cost. Dense bases would add $(2KR + 2RN)/(2KN) = R/N + R/K$, or 17.14% at $K = 5120$, $N = 7168$, and $R = 512$. The structured path costs

$$F_{\text{structured}} = K \log_2 k_b + R \log_2 R + (T-1)R + K + 2R + 2N \tag{4}$$

Algorithm 1 NoisyGEMM linear-layer computation without materialized weight perturbations

Require: X, shared W, coefficient table G, token-to-slot map s, fixed basis metadata
1: Compute $H = XA$ using blockwise signed Hadamard transforms
2: For each token m, form $H_m \leftarrow H_m \odot G_{s(m)}/\sqrt{K}$
3: Apply the output Hadamard transform; retain an $M \times R$ buffer U
4: Compute $XW^\top$ with weight tiles shared across token rows
5: Add the corresponding signed expansion of U in each output tile's epilogue
6: Return Y; release the transform buffer after all output tiles have read it

FLOPs per token for the two transforms, block aggregation, input signs, coefficient scaling, and output signs and addition. With $R = k_b = 512$ this is 75,776 FLOPs against 73,400,320 for the base projection, about 0.10%.

D Hadamard Basis and Coefficient-to-Weight Isometry

Let H_n be the unnormalized Sylvester–Hadamard matrix, so $H_n H_n^\top = nI$. Choose power-of-two k_b and R, with $R \le k_b$, $k_b \mid K$, and $R \mid N$. Let S select R distinct columns of H_{k_b}. For input sign vectors $d_1^{(\ell)} \in \{\pm 1\}^{k_b}$ and output sign vectors $d_2^{(\ell)} \in \{\pm 1\}^R$, define

$$A = \begin{bmatrix} \operatorname{diag}(d_1^{(1)}) H_{k_b}[:, S] \\ \vdots \\ \operatorname{diag}(d_1^{(K/k_b)}) H_{k_b}[:, S] \end{bmatrix}, \qquad B = \begin{bmatrix} H_R \operatorname{diag}(d_2^{(1)}) & \cdots & H_R \operatorname{diag}(d_2^{(N/R)}) \end{bmatrix}. \tag{5}$$

Sign flips preserve inner products, and each input block contributes $k_b I_R$ to $A^\top A$. Thus $A^\top A = K I_R$. Each output block contributes $R I_R$ to $BB^\top$, giving $BB^\top = N I_R$. Define the coefficient-to-weight map

$$L(g) = \frac{(A \operatorname{diag}(g) B)^\top}{\sqrt{K}}. \tag{6}$$

For $c = g - h$ and $C = \operatorname{diag}(c)$,

$$\|L(g) - L(h)\|_F^2 = K^{-1} \operatorname{tr}(B^\top C A^\top A C B) \tag{7}$$

$$= \operatorname{tr}(C^2 B B^\top) = N \|c\|_2^2. \tag{8}$$

By polarization, $\langle L(u), L(v) \rangle_F = N \langle u, v \rangle$. Thus the weight-update matrices $E_j = L(e_j)/\sqrt{N}$ form an orthonormal basis of $\operatorname{range}(L)$. For $\varepsilon \sim \mathcal{N}(0, I_R)$,

$$L(\sigma\varepsilon) = \sigma\sqrt{N} \sum_{j=1}^{R} \varepsilon_j E_j, \qquad \mathbb{E}\|L(\sigma\varepsilon)\|_F^2 = N\sigma^2 R.$$

Consequently, isotropic coefficient noise remains isotropic within this subspace, while coordinate scales d_j induce variances $N\sigma^2 d_j^2$ along its orthogonal directions. These identities hold for each adapted weight matrix; its output dimension N determines the scale factor.

E Parallel Execution

A fixed global basis. The sign vectors, the selected Hadamard columns, and k_b are part of the model definition. Every rank derives them for the global layer widths from the basis seed and the layer name and slices its shard by global offset, so changing the sharding plan does not redraw the basis.

Column-parallel layers. When the output dimension is sharded, every rank computes the full intermediate u and expands only its own output coordinates, $\Delta y_n = u_{(n+n_{\text{off}}) \bmod R}\, d_{2,\, n+n_{\text{off}}}$.

Row-parallel layers. When the input dimension is sharded, each rank projects only its local input blocks (k_b must divide the shard width) and adds the resulting partial correction to its partial base output. By linearity, $\sum_p((x_pA_p) \odot g)B = ((\sum_p x_pA_p) \odot g)B$, so the existing all-reduce also sums the corrections, with no extra communication.

Multi-node training. A controller owns the data, samples prompts and seeds, scores completions, and runs the optimizer; inference nodes hold the frozen weights, the basis, and their KV caches. Per step, the controller sends the prompts tagged with seeds, plus the search center and per-coefficient scales ($2D$ values, 344 KB for Qwen3-1.7B with $D = 43{,}008$), and nodes return completions. A single bf16 all-reduce of Qwen3-1.7B's gradients moves about 3.4 GB.

F RLVR ADJUSTMENTS

Our RLVR optimizer uses two adjustments to separable CMA-ES that address properties of verifiable rewards.

Group-relative fitness. A verifiable reward is nearly binary, and question difficulty spans the whole range from always solved to never solved. If each candidate is scored on its own random questions, its fitness mostly measures how easy its questions were, while a perturbation at a usable σ moves the solve rate by a few points; ranking such fitness selects easy draws rather than good perturbations. Let P be the number of questions assigned to each candidate and n the number of completions per assigned question. We sample $Q < \lambda P$ questions per step and tile them across candidates, so each question is answered by multiple candidates. With $\mathcal{G}_q$ the rollouts on question q, each rollout is scored against its group mean and the residuals are averaged per candidate:

$$b_q = \frac{1}{|\mathcal{G}_q|} \sum_{(i,k)\in\mathcal{G}_q} r_{q,i,k}, \qquad F_i = \frac{1}{Pn} \sum_{(q,k)\text{ assigned to } i} (r_{q,i,k} - b_q)\,. \tag{9}$$

We subtract the per-question population mean to form group-relative fitness. Unlike GRPO's group of samples from one policy (Shao et al., 2024), this group spans different perturbed policies. Questions with identical rewards across candidates contribute zero to the fitness signal.

Score-weighted update with momentum. Because F_i is meaningful in magnitude and not only in rank, the mean moves by a zero-mean, score-weighted step over the whole population, with bias-corrected momentum. For nonzero fitness variance,

$$w_i = \frac{F_i - \bar{F}}{\lambda\,\mathrm{std}(F)}, \quad \Delta_t = \sum_{i=1}^{\lambda} w_i\,\sigma_t\,d_t \odot \varepsilon_{t,i}, \quad v_t = \beta v_{t-1} + (1-\beta)\Delta_t, \quad m_{t+1} = m_t + \eta\,\frac{v_t}{1-\beta^t}. \tag{10}$$

When all fitness values are equal, the score-weighted direction is zero; we define $w_i = 0$ in this case. Below-average candidates push the mean away from their perturbations instead of being ignored, and the $1/\mathrm{std}(F)$ factor makes the step invariant to the reward scale. Momentum averages the sampled directions across steps without additional rollouts. Bias correction removes the initialization bias from $v_0 = 0$. σ_t and d_t still adapt as in Appendix G. Both adjustments cost $O(D)$ state and need only λ scalar fitness values; Δ_t is the same seed-regenerated weighted sum as the CMA-ES moments.

G SEPARABLE CMA-ES UPDATES

The separable CMA-ES variant ranks the candidates, selects the top μ with log-rank weights b_k ($\sum_k b_k = 1$, $\mu_{\text{eff}} = (\sum_k b_k^2)^{-1}$), and uses the weighted noise moments

$$z_t = \sum_{k=1}^{\mu} b_k \varepsilon_{t,i_k}, \qquad q_t = \sum_{k=1}^{\mu} b_k \varepsilon_{t,i_k}^{\odot 2}$$

to update the search center, $m_{t+1} = m_t + c_m\,\sigma_t\,d_t \odot z_t$, and the step size and per-coefficient scales:

$$p^\sigma_{t+1} = (1-c_\sigma)p^\sigma_t + \sqrt{c_\sigma(2-c_\sigma)\mu_{\text{eff}}}\,z_t, \tag{11}$$

$$\sigma_{t+1} = \text{clip}\left(\sigma_t \exp\left[\frac{c_\sigma}{d_\sigma}\left(\frac{\|p^\sigma_{t+1}\|_2}{\chi_D} - 1\right)\right], \sigma_{\min}, \sigma_{\max}\right), \tag{12}$$

$$h_t = \mathbf{1}\left\{\frac{\|p^\sigma_{t+1}\|_2}{\sqrt{1-(1-c_\sigma)^{2t}}} < \left(1.4 + \frac{2}{D+1}\right)\chi_D\right\}, \tag{13}$$

$$p^c_{t+1} = (1-c_c)p^c_t + h_t\sqrt{c_c(2-c_c)\mu_{\text{eff}}}\,(d_t \odot z_t), \tag{14}$$

$$C_{t+1} = \text{clip}\Big([1 - c_1 - c_\mu + c_1(1-h_t)c_c(2-c_c)]C_t + c_1(p^c_{t+1})^{\odot 2} + c_\mu(C_t \odot q_t), C_{\min}, C_{\max}\Big), \tag{15}$$

$$d_{t+1} = \sqrt{C_{t+1}}. \tag{16}$$

Here $C_t = d_t^{\odot 2}$, $\chi_D = \mathbb{E}\|\mathcal{N}(0, I_D)\|_2$, and clipping acts elementwise. The paths start at $p^\sigma_1 = p^c_1 = 0$ and $t = 1$ denotes the first update. The diagonal covariance update uses the standard path-length gate and its covariance correction (Hansen, 2016; Ros & Hansen, 2008). The step-size path uses whitened noise z_t, whereas the covariance path and second moment use the scaled steps $d_t \odot z_t$ and $C_t \odot q_t$. Because each ε_{t,i_k} is a function of its seed, z_t and q_t are accumulated tile by tile by regenerating the noise, without storing the $\lambda \times D$ population. With the adjustments of Appendix F, the search center follows Eq. (10) instead of the rank-weighted step above.

Efficient per-coefficient scales. EGGROLL perturbs a weight with rank-r factors, $\Delta_b = P_b Q_b^\top/\sqrt{r}$, and applies it as $(xQ_b)P_b^\top/\sqrt{r}$ without forming Δ_b. A per-parameter scale S, as separable CMA-ES would learn over the weights, breaks this factorization: $[x(S \odot \Delta_b)^\top]_i = \frac{1}{\sqrt{r}}\sum_j (P_b)_{ij}\sum_k x_k S_{ik}(Q_b)_{kj}$ couples the output index i with the input index k and destroys the two-product low-rank evaluation path for a general S. A separable scale $S = ab^\top$ can instead be absorbed into the factors. In NOISYGEMM, the scale acts on the coefficients, $\Delta W_b^\top = A\,\text{diag}(\sigma_t d_t \odot \varepsilon_{t,b})\,B/\sqrt{K}$, so it requires an elementwise product on an R-vector before expansion.

H EVALUATION DETAILS

Main quality experiment (Table 2). The main quality setup uses one node with eight A100 80 GB GPUs. Math training data comes from OpenR1-Math-220k, DAPO-Math-17k, and NuminaMath-CoT, and coding data from the Nemotron and Eurus-2-RL coding sets. All methods share the prompts, verifier reward, and maximum completion length of 16k tokens; NOISYGEMM uses a separable CMA-ES variant with $\lambda = 2048$ and $R = 512$.

Convergence and ablations. Figures 3a and 4 use Qwen3-4B on AIME 2025 with 8192 rollouts per step. The adapted run uses $\lambda = 512$ and 16 rollouts per candidate. The population sweep varies λ while keeping this product fixed. The fixed-scale runs use isotropic perturbations with $d = \mathbf{1}$ and $\sigma \in \{0.001, 0.005\}$; the EGGROLL ablation uses ranks 1 and 4. Ablation gains are averaged over steps 90–120, while Figure 3a shows steps 0–100. These learning curves start at 35.8% AIME 2025 accuracy and are reported separately from the main quality runs in Table 2.

Training throughput (Figure 3b, c). Each generation rolls out Countdown prompts, 4 tasks per candidate and at most 512 tokens, on Qwen3-1.7B on one A100; the rate includes perturbation setup and the update. The MeZO-style sequential baseline rolls out 32 candidates and is extrapolated linearly. EGGROLL's rank-16 adapters are capped at 16 GB, so 357 of them fit at $\lambda = 512$. We tag candidates through vLLM's LoRA slots, capped at 256 resident slots while evaluating populations of up to 1024 candidates. Generation 0 is discarded as warm-up; plotted rates average the remaining generations. Runs use bf16 weights, sampling temperature 1, top-$p = 1$, and GPU memory utilization 0.85.

Live-serving replay (Figure 6). The replay uses Llama-3.1-8B-Instruct, 2.5% of requests from BurstGPT day 9, and 144× time compression. A day of arrivals therefore spans ten minutes of wall-clock time; the plot includes the final drain period. Training completions have 64 output tokens. Throughput counts output tokens for both live and training requests. Each request's tokens are distributed uniformly over its submit-to-finish interval. Latency is the P99 of per-request mean TPOT, grouped by arrival time in 10-minute trace buckets (about 4.17 seconds of wall-clock time). The target for this metric is 500 ms. Token counts are 240,851 live outputs and 1,103,488 training outputs; they do not include prompt tokens.

I KERNEL MICROBENCHMARKS

Protocol. We time one perturbed linear layer, $y_i = x_i(W + \sigma\,\Delta W_{c(i)})^\top$, for M tokens spread evenly over λ candidates, in bf16 with CUDA graphs on an A100-SXM4-80GB, reporting the median of three runs. *Full-rank materialization* stores λ perturbed weight copies and runs one batched GEMM; *low-rank* (EGGROLL-style) stores rank-r factors per candidate ($r = 16$ unless stated) and adds $(xA_b)B_b$; NOISYGEMM stores $R = 512$ coefficients per candidate. These kernel baselines measure evaluation paths rather than full training algorithms. Both baselines receive tokens pre-sorted by candidate and exclude noise generation, which favors them.

Results. NOISYGEMM costs 1.29×–1.31× cuBLAS from $\lambda = 256$ to 2048, while full-rank perturbations cost 15×–116× and rank-16 factors 1.71×–3.51×. Low-rank cost also grows with rank, from 2.02× at $r = 16$ to 9.99× at $r = 512$, while NOISYGEMM costs 1.15×–1.21× from $R = 128$ to 1024 (Figure 9c). Across Qwen3 model sizes, NOISYGEMM costs 1.12×–1.33×, rank-16 factors 1.55×–2.17×, and full-rank perturbations 22×–41× (Figure 9b).

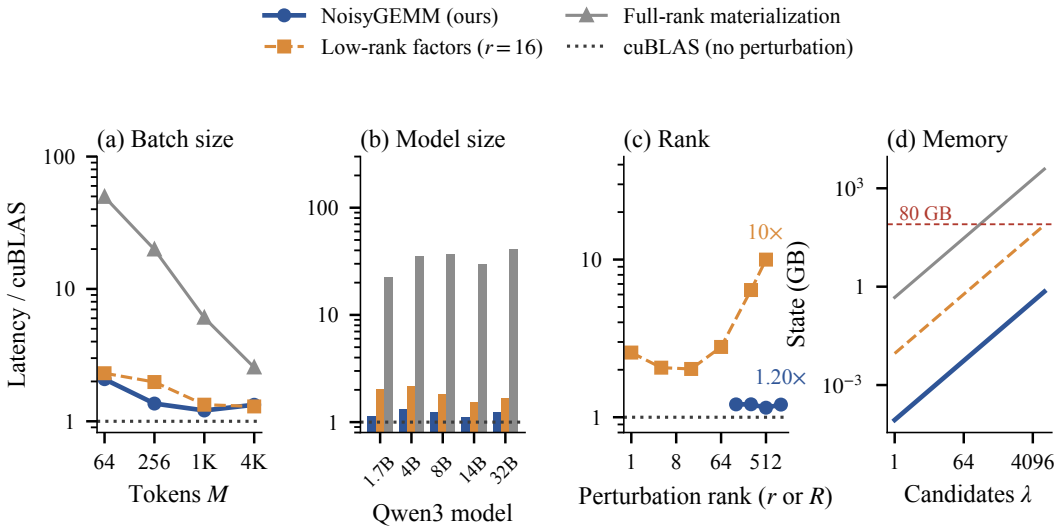


Figure 9: **Kernel microbenchmarks on an A100-SXM4-80GB. (a)** Latency of one perturbed Qwen3-1.7B query/key/value layer relative to cuBLAS as the number of tokens grows, with $\lambda = 64$. **(b)** The same across Qwen3 model sizes with $M = 1024$ and $\lambda = 256$. **(c)** Cost versus perturbation rank: low-rank factors of rank r and NOISYGEMM with R coefficients ($M = 1024$, $\lambda = 256$). **(d)** Perturbation state for Qwen3-1.7B with adapted query, key, and value projections in bf16, by exact count: a weight-sized tensor per candidate (full-rank), rank-16 factors per candidate (low-rank), or R coefficients per adapted matrix (NOISYGEMM); the dashed line is the 80 GB of an A100.