

Randomized Gradient Compression for Memory-Efficient Reinforcement Learning

Ben Zhao
Iluvatar Labs Inc.
ben@iluvatarlabs.com

13 October 2026
Last updated 16 October 2026 (version 2)*

Abstract

Reinforcement learning for language models needs memory both to generate responses and to optimize the policy. Randomized Gradient Compression (RGC) trains a model’s original weights from a compressed gradient: each selected layer forms its weight gradient through a random orthogonal projection of its input, and a moment-free SGD rule applies the reconstructed update in row blocks, so neither a dense weight gradient nor optimizer moments are ever stored. On Qwen3-1.7B-Base trained with GRPO on GSM8K, RGC with per-matrix normalized SGD comes within 0.4–1.7 test-accuracy points of LoRA at ranks 64, 128 and 256 and within 1.7 points of dense AdamW, while using 63–65% less policy-training memory and 38–40% less whole-loop memory than dense AdamW. A state-accounting model, fixed before training, predicts RGC’s training-memory gap to LoRA at each rank; measured peaks match it to within about 0.1 GiB, with RGC 8.4% below LoRA at rank 256 and LoRA lower at ranks 64 and 128. Normalized SGD outperforms plain SGD at every rank. All results come from a single training seed.

1 Introduction

Reinforcement learning (RL) is now a standard way to improve the reasoning of language models, but training the policy is memory-hungry: dense optimization keeps a gradient and optimizer moments for every trained weight, on top of the model, its activations and the generation cache. Adafactor reduces optimizer state by factoring second-moment estimates [19], MeZO avoids backpropagation with zeroth-order estimates [8], and low-rank adaptation (LoRA) freezes the base weights and trains a low-rank additive term [4], which remains effective at small ranks in RL [15]. Another family changes the pretrained matrices directly but shrinks what the optimizer sees or sends: PowerSGD transmits rank-limited gradient approximations [20], GaLore runs the optimizer inside a gradient subspace recomputed by singular value decomposition [25], Flora uses random projections that change over training [2], and GoLore draws its projections uniformly from orthonormal frames [3]. CompAct stores randomly projected layer inputs for the backward pass [17], and LoRA-FA freezes LoRA’s down-projection so that only the projected input is stored [24].

Randomized Gradient Compression (RGC) places a random orthogonal projection inside the weight-gradient computation of every selected layer, so that gradient exists only in compressed form, and updates the original weights with one of two moment-free SGD rules: plain SGD, or a per-matrix normalized rule in the spirit of LARS [22]. RGC’s persistent training state is fixed by the layer shapes and the rank, so where its training memory crosses below LoRA’s can be computed before training. Our contributions are:

- an original-weight RL update that reconstructs each layer’s gradient from a redrawn, uniformly random orthogonal projection, which is unbiased and has the smallest worst-case error of any unbiased rank- r projection, and applies it in 128-row blocks with per-matrix normalized SGD, with no dense gradient and no optimizer moments;

*Version 2 replaces version 1 with a new study.

- a state-accounting model that predicts, before training, where RGC’s training memory crosses below LoRA’s as rank grows, and measurements that match its predicted differences to within about 0.1 GiB;
- a comparison on Qwen3-1.7B-Base and GSM8K against dense AdamW and LoRA at three ranks, in which normalized RGC stays within 0.4–1.7 test points of LoRA and dense AdamW while using 63–65% less training memory than dense AdamW.

2 Method

Compressed accumulation. For a selected matrix $W_\ell \in \mathbb{R}^{d_\ell \times k_\ell}$, microbatch m has input $X_{\ell m}$, output $Y_{\ell m} = X_{\ell m} W_\ell^\top$ and output derivative $D_{\ell m} = \partial \mathcal{L} / \partial Y_{\ell m}$, where the loss $\mathcal{L}$ includes the full-batch token normalization. RGC accumulates

$$C_\ell = \sum_m D_{\ell m}^\top (X_{\ell m} P_\ell), \quad H_\ell = C_\ell P_\ell^\top. \quad (1)$$

As in CompAct [17], each selected layer saves the projected input $X_{\ell m} P_\ell$ in place of $X_{\ell m}$ for the backward pass, while its output and the derivative $D_{\ell m} W_\ell$ passed to earlier layers are unchanged. Because W_ℓ and P_ℓ change only between steps, $C_\ell = G_\ell P_\ell$, where $G_\ell = \sum_m D_{\ell m}^\top X_{\ell m}$ is the dense weight gradient, which is never formed. The FP32 accumulator C_ℓ is $d_\ell \times r$ and the projection P_ℓ is $k_\ell \times r$. H_ℓ is only ever formed in blocks of 128 output rows, so no dense weight gradient or dense update is allocated during training.

Random orthogonal projection. For each selected matrix, draw a standard Gaussian $k_\ell \times r$ matrix in FP32, compute its reduced QR factorization, correct the column signs of Q_ℓ by the signs of R_ℓ ’s diagonal [9, 13], and set

$$P_\ell = \sqrt{k_\ell / r} Q_\ell, \quad Q_\ell^\top Q_\ell = I_r. \quad (2)$$

This gives a uniformly random orthonormal frame, as in GoLore [3]. RGC stores P_ℓ in BF16 and releases the QR workspace before drawing the next layer’s projection. Projections come from a dedicated random-number stream (seed 1000042) that response sampling does not use, and layers draw independently. After every applied update a new projection is drawn for the next step; a step that applies no update keeps its unused projection.

Two moment-free update rules. With warmup $w_t = \min(t/5, 1)$ for scheduled step $t \geq 1$, plain SGD applies

$$W_\ell \leftarrow W_\ell - w_t \eta_r H_\ell. \quad (3)$$

The normalized rule rescales each matrix’s update by that matrix’s own weight-to-update norm ratio:

$$Z_\ell = \frac{\|W_\ell\|_F}{\|H_\ell\|_F + 10^{-12}} H_\ell, \quad W_\ell \leftarrow W_\ell - w_t \tau_r Z_\ell. \quad (4)$$

This is LARS-style layerwise scaling [22] without momentum or weight decay. Plain SGD keeps the estimated gradient’s relative magnitudes across matrices; the normalized rule moves every matrix by the same fraction $w_t \tau_r$ of its own norm. A zero H_ℓ leaves its matrix unchanged. Norms are reduced blockwise in FP64 and fixed before the update is applied; reconstruction and weight addition use FP32. Neither rule stores any optimizer state.

Estimator properties. For a fixed gradient $G \in \mathbb{R}^{d \times k}$ independent of a freshly drawn projection, $A = QQ^\top$ is a uniformly random rank- r orthogonal projector with $A^2 = A$ and $\mathbb{E}[A] = (r/k)I_k$. Hence

$$\mathbb{E}[PP^\top] = I_k, \quad \mathbb{E}[GPP^\top \mid G] = G, \quad (5)$$

$$\mathbb{E}\left[\|GPP^\top - G\|_F^2 \mid G\right] = \left(\frac{k}{r} - 1\right) \|G\|_F^2. \quad (6)$$

Expanding $((k/r)A - I_k)^2$ and taking expectations gives Equation 6. No unbiased reconstruction of this form does better in the worst case: for $M = PP^\top$, which is positive semidefinite with rank at most r and $\mathbb{E}[M] = I_k$, the inequality $\text{tr}(M^2) \geq (\text{tr } M)^2/r$ and Jensen’s inequality give $\text{tr } \mathbb{E}[M^2] \geq k^2/r$, so $\mathbb{E}[M^2]$ has an eigenvalue of at least k/r and some G incurs error at least $(k/r - 1) \|G\|_F^2$. The scaled orthogonal frame attains this bound for every G . Like randomized automatic differentiation [10], the estimator trades variance for memory. These identities hold in exact arithmetic; BF16 storage of P_ℓ and BF16 products make them approximate in the implementation, and the normalized rule’s nonlinear scaling does not inherit unbiasedness. The rank of each update is at most r , so $\text{rank}(W_T - W_0) \leq \min(d, k, Tr)$ after T applied updates: as in Flora [2], redrawing the projection lets rank- r updates add up to a higher-rank change, an effect ReLoRA obtains through periodic restarts [7].

Memory accounting. Let $N_W = \sum_\ell d_\ell k_\ell$ over the selected matrices. Above a common all-BF16 base, the persistent training state in bytes is

$$S_{\text{RGC}}(r) = 2N_W + \sum_\ell (4d_\ell r + 2k_\ell r), \quad (7)$$

$$S_{\text{LoRA}}(r) = 16r \sum_\ell (d_\ell + k_\ell), \quad (8)$$

$$S_{\text{dense}} = 14N_W. \quad (9)$$

For RGC, $2N_W$ is the premium for keeping the trained weights in FP32 and the remaining terms are the FP32 accumulators and BF16 projections. For LoRA, the factor 16 covers FP32 adapter weights, their gradients and two Adam moments; dense AdamW adds FP32 gradients and two moments for every trained weight. Activations and transient buffers are excluded. For Qwen3-1.7B-Base, $N_W = 1,409,286,144$, so the FP32 premium is 2.625 GiB, RGC’s compressed state costs 3,325,952 bytes per rank and LoRA’s adapter state 17,432,576 bytes per rank. Equating Equations 7 and 8 predicts that RGC at ranks 64, 128 and 256 matches LoRA’s state at LoRA ranks of about 174, 186 and 211: LoRA should be smaller at ranks 64 and 128, and RGC smaller at rank 256.

3 Experimental setup

Model, data and reward. We train Qwen3-1.7B-Base (revision ea980cb0a6c2ae4b936e82123acc929f1cec04c1) [21, 14] on GSM8K (main, revision 740312add88f781978c0658806c59bc2815b9866) [1, 11], split into 7,217 training and 256 development questions, with all 1,319 test questions held out. The chat template shipped with the model formats a system message, “Please reason step by step, and put your final answer within `\boxed{\}`.”, and the question. Generation stops on `<|endoftext|>` or `<|im_end|>`. The reward is 1 when Math-Verify 0.8.0 [6] judges the boxed answer equal to the reference and 0 otherwise, including answers that are missing, cannot be parsed or are truncated at the 1,024-token completion cap; truncated responses are excluded from the loss. Evaluation is greedy with the same cap and scoring.

Objective. Each step samples four responses for each of 16 prompts and accumulates over 64 microbatches of one response each. Following GRPO [18, 5], response i to question q has advantage $A_{qi} = (R_{qi} - \bar{R}_q)/(s_q + 10^{-4})$, with s_q the sample standard deviation of the group’s rewards. With usable-token mask M_{qij} and $N_+ = \max(\sum_{qij} M_{qij}, 1)$,

$$\mathcal{L}(\theta) = -\frac{1}{N_+} \sum_{qij} M_{qij} A_{qi} \exp(\log \pi_\theta(o_{qij} \mid q, o_{qi, < j}) - \text{sg}[\log \pi_\theta(o_{qij} \mid q, o_{qi, < j})]), \quad (10)$$

where `sg` stops the gradient and the normalization over all usable tokens in the rollout follows DAPO [23]. Each fresh batch is used for exactly one update, so the ratio is one in value and PPO [16] clipping never acts. There is no reference KL, entropy bonus or gradient clipping. A batch in which every unmasked response has zero advantage applies no update to the weights, the Adam moments or the projections. Sampling uses temperature 1 and top- p 1 with no top- k filtering and a 3,072-token prompt cap; every method warms up over five steps and then holds its scale constant.

Baselines and precision. All methods train the query, key, value and output projections and the MLP gate, up and down projections of all 28 blocks; everything else is frozen. Dense and RGC keep these weights in FP32. LoRA keeps a BF16 base and FP32 adapters with standard PEFT initialization, no dropout and $\alpha/r = 0.5$. Dense and LoRA use single-tensor AdamW [12] with betas (0.9, 0.95), $\epsilon = 10^{-8}$ and no weight decay. Computation uses BF16, scaled dot-product attention and non-reentrant activation checkpointing, with PyTorch 2.9.1, Transformers 4.57.1, TRL 0.24.0 and PEFT 0.17.1. Every run occupies a single RTX PRO 6000 Blackwell GPU; generation uses the key/value cache with weight-cast caching disabled for every method, and RGC’s accumulators and projections stay resident throughout.

Step-size calibration. RGC’s base scales are calibrated on the untrained model rather than tuned freely. Eight training-only rollout batches (seed 1042) are saved and replayed at unchanged weights. For each batch b , the dense gradient g_b defines the first AdamW step from zero moments at learning rate 10^{-6} ,

$$u_b = -10^{-6} \frac{g_b}{|g_b| + 10^{-8}}, \quad \delta_0 = \text{median}_b \text{RMS}(u_b) \approx 9.89 \times 10^{-7}, \quad (11)$$

with elementwise division and RMS taken over all selected weights. Replaying the same batches through RGC at rank r gives update directions $H_{b,r}$ and $Z_{b,r}$, and the base scales (Table 1) are

$$\eta_{0,r} = \frac{\delta_0}{\text{median}_b \text{RMS}(H_{b,r})}, \quad \tau_{0,r} = \frac{\delta_0}{\text{median}_b \text{RMS}(Z_{b,r})}. \quad (12)$$

Each rule tries its base scale, which matches the size of a first AdamW step at learning rate 10^{-6} , and three times that scale; all scales were fixed before training.

Table 1: Calibrated RGC base scales. $\tau_{0,r}$ does not depend on the rank because $\|Z_\ell\|_F = \|W_\ell\|_F$.

Rank	Plain SGD $\eta_{0,r}$	Normalized SGD $\tau_{0,r}$
64	4.06×10^{-3}	2.80×10^{-5}
128	6.06×10^{-3}	2.80×10^{-5}
256	8.75×10^{-3}	2.80×10^{-5}

Grid and selection. Ten configurations each train two candidates for 200 steps from the same base checkpoint with training seed 42: dense AdamW (learning rates 3×10^{-6} and 10^{-5}), LoRA at ranks 64, 128 and 256 (3×10^{-5} and 10^{-4}), and plain and normalized RGC at ranks 64, 128 and 256 (the calibrated scale and three times it). Development accuracy is measured every 50 steps. In each configuration the candidate with the higher development accuracy at step 200 is selected, with ties going to the lower scale, and all ten selections were fixed before any test evaluation. Four interrupted candidates were rerun unchanged from the base checkpoint. Training and whole-loop memory are the maximum allocated CUDA peaks over each run’s policy-training phases and over its complete steps, generation included. For each RGC configuration, a paired bootstrap over test questions (10,000 resamples, seed 1043) gives a 95% interval for its accuracy difference from same-rank LoRA and from dense AdamW.

4 Results

Table 2 and Figure 1 give the full quality–memory comparison; Table 4 gives the paired differences.

Quality. Training lifts test accuracy from 64.90% to between 79.98% and 83.78% in every configuration. Normalized RGC reaches 83.24%, 82.03% and 82.34% at ranks 64, 128 and 256, within 0.38, 1.74 and 1.29 points of LoRA at the same rank and within 0.45, 1.67 and 1.36 points of dense AdamW (83.70%). LoRA128 is the most accurate configuration at 83.78%, and LoRA64 uses the least memory of any configuration. None of normalized RGC’s six paired 95% intervals against LoRA or dense AdamW excludes zero (Table 4). Normalized SGD beats plain SGD at every rank, by 12, 27 and 11 test questions.

Table 2: Test accuracy and memory for the selected candidate of every configuration. Train and Loop are allocated peaks for policy training and the full loop; Reserved is the whole-loop allocator-reserved peak. GiB is 2^{30} bytes. Loop time excludes setup, evaluation and checkpointing and is descriptive.

Method	Rank	Test (%)	Train (GiB)	Loop (GiB)	Reserved (GiB)	Loop (min)
Untrained base	–	64.90	–	–	–	–
Dense AdamW	–	83.70	24.66	25.37	50.16	133.03
LoRA + AdamW	64	83.62	6.75	13.16	41.77	171.70
LoRA + AdamW	128	83.78	7.78	13.93	42.66	209.68
LoRA + AdamW	256	83.62	9.96	15.12	42.33	161.41
RGC plain SGD	64	82.34	8.53	15.10	50.33	167.24
RGC plain SGD	128	79.98	8.73	15.30	50.43	164.01
RGC plain SGD	256	81.50	9.13	15.69	50.85	161.64
RGC normalized SGD	64	83.24	8.53	15.10	50.33	152.43
RGC normalized SGD	128	82.03	8.73	15.11	50.43	164.92
RGC normalized SGD	256	82.34	9.13	15.69	50.85	192.09

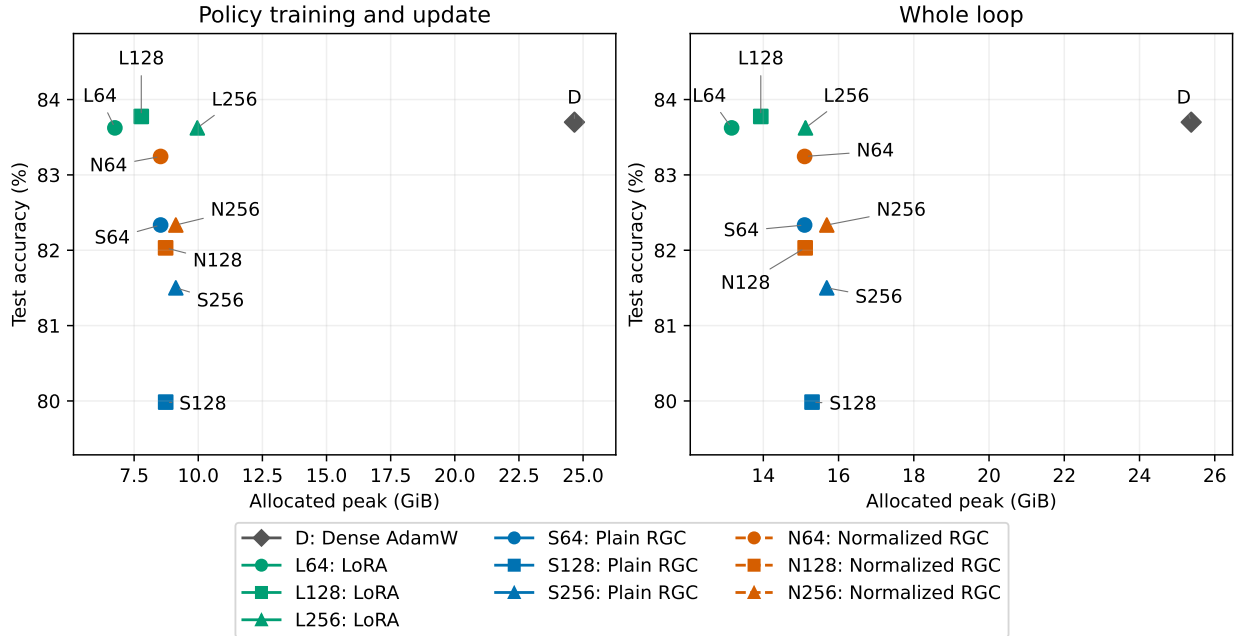


Figure 1: Test accuracy versus allocated peak memory for policy training (left) and the full loop (right). D is dense AdamW, L LoRA, S plain RGC and N normalized RGC; numbers give the rank.

Memory against dense training. Normalized RGC cuts the policy-training peak from 24.66 GiB under dense AdamW to 8.53–9.13 GiB, a 63–65% reduction, and the whole-loop peak from 25.37 GiB to 15.10–15.69 GiB, a 38–40% reduction, while staying within 0.45–1.67 points of dense AdamW’s accuracy.

The predicted crossover with LoRA. Table 3 compares the state-accounting prediction from Section 2 with the measured training peaks. The predicted differences match the measurements to within about 0.1 GiB at every rank: LoRA’s training peak is lower at ranks 64 and 128, and RGC’s is lower at rank 256, by 0.84 GiB (8.4%). Over the full loop, where generation sets the peak, RGC stays above LoRA at all three ranks, by 0.57 GiB at rank 256. Allocator-reserved peaks, which include cached blocks, are 50.3–50.8 GiB for RGC, 41.8–42.7 GiB for LoRA and 50.2 GiB for dense AdamW; the reductions above refer to allocated memory.

Table 3: RGC minus same-rank LoRA training memory: the persistent-state prediction (Section 2) versus the measured difference in allocated policy-training peaks, in GiB.

Rank	Predicted	Measured, plain SGD	Measured, normalized SGD
64	+1.784	+1.783	+1.783
128	+0.943	+0.951	+0.948
256	−0.738	−0.838	−0.838

Table 4: RGC accuracy minus same-rank LoRA and minus dense AdamW (D), in test percentage points, with paired-question 95% bootstrap intervals that hold the selected checkpoints fixed.

RGC row	Δ LoRA [95% CI]	Δ D [95% CI]
Plain 64	−1.29 [−3.26, +0.68]	−1.36 [−3.34, +0.61]
Plain 128	−3.79 [−5.84, −1.74]	−3.71 [−5.76, −1.74]
Plain 256	−2.12 [−4.17, −0.15]	−2.20 [−4.17, −0.23]
Normalized 64	−0.38 [−2.27, +1.59]	−0.45 [−2.43, +1.52]
Normalized 128	−1.74 [−3.79, +0.30]	−1.67 [−3.71, +0.38]
Normalized 256	−1.29 [−3.34, +0.68]	−1.36 [−3.26, +0.53]

Development versus test. Development rankings did not reliably predict test rankings. Plain RGC64 had the best development score of any candidate (238/256) yet trails LoRA64 on test, and normalized RGC256 led LoRA256 on development (235 versus 228) but trails it on test. Appendix A lists every candidate’s development trajectory, including a dense AdamW run at 10^{-5} whose accuracy fell late as answers hit the length cap, and a plain RGC256 run at three times its base scale that collapsed into length-capped answers.

5 Discussion

RGC trains every attention and MLP projection of a 1.7B model with RL without storing a dense gradient or any optimizer moments. It comes within 0.4–1.7 points of LoRA and dense AdamW using 35–37% of dense AdamW’s training memory, and the accounting model predicted its training-memory gaps to LoRA, to within about 0.1 GiB, before training began.

That model also identifies the term that decides the LoRA comparison. RGC stores the weights it trains in FP32, a 2.6 GiB premium for this model that LoRA avoids by freezing a BF16 base. The premium places the crossover between LoRA ranks 174 and 211 here. Storing the trained weights in BF16 would remove it and, by the same accounting, put RGC’s training state below LoRA’s at every tested rank; whether learning survives BF16 updates of this size is the open question and the most direct next step. The update rule matters as well: moving every matrix by the same fraction of its norm beat plain SGD on test at all three ranks.

The study covers one model, one task, a 200-step horizon and a 1,024-token cap; additional seeds will narrow the paired intervals in Table 4.

6 Conclusion

Randomized Gradient Compression makes original-weight RL fit in roughly a third of dense AdamW’s training memory at accuracy within 0.4–1.7 points of both dense AdamW and LoRA, and where its training memory crosses below LoRA’s can be predicted before training.

References

- [1] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- [2] Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.
- [3] Yutong He, Pengrui Li, Yipeng Hu, Chuyan Chen, and Kun Yuan. Subspace optimization for large language models with convergence guarantees. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 22468–22522. PMLR, 2025.
- [4] 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.
- [5] Hugging Face TRL contributors. GRPO trainer. TRL version 0.24.0 documentation and source, 2025. Implementation: https://github.com/huggingface/trl/blob/v0.24.0/trl/trainer/grpo_trainer.py.
- [6] Hynek Kydlíček. Math-Verify: Math verification library. Version 0.8.0, 2025. Source repository: github.com/huggingface/Math-Verify.
- [7] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. ReLoRA: High-rank training through low-rank updates. In *International Conference on Learning Representations*, 2024.
- [8] 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.
- [9] Francesco Mezzadri. How to generate random matrices from the classical compact groups. *Notices of the American Mathematical Society*, 54(5):592–604, 2007.
- [10] Deniz Oktay, Nick McGreivy, Joshua Aduol, Alex Beatson, and Ryan P. Adams. Randomized automatic differentiation. In *International Conference on Learning Representations*, 2021.
- [11] OpenAI. GSM8K: Grade school math 8k, main configuration. Hugging Face dataset repository, 2021. Revision 740312add88f781978c0658806c59bc2815b9866.
- [12] PyTorch contributors. AdamW. PyTorch 2.9 documentation, 2025.
- [13] PyTorch contributors. `torch.linalg.qr`. PyTorch 2.9 documentation, 2025.
- [14] Qwen Team. Qwen3-1.7B-Base: Model, configuration and tokenizer. Hugging Face model repository, 2025. Revision ea980cb0a6c2ae4b936e82123acc929f1cec04c1.
- [15] John Schulman and Thinking Machines Lab. LoRA without regret. Thinking Machines Lab, September 2025.
- [16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [17] Yara Shamshoum, Nitzan Hodos, Yuval Sieradzki, and Assaf Schuster. CompAct: Compressed activations for memory-efficient LLM training. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1511–1524, 2025.
- [18] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

- [19] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4596–4604, 2018.
- [20] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, 2019.
- [21] An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [22] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.
- [23] Qiyang Yu et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- [24] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. LoRA-FA: Efficient and effective low rank representation fine-tuning. *arXiv preprint arXiv:2308.03303*, 2023.
- [25] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

A Development trajectories

Table 5: Development results for all twenty candidates. Each cell gives correct answers / truncated answers / mean completion tokens over 256 questions at the stated step. D is dense AdamW, L LoRA, S plain RGC and N normalized RGC; suffix 1/2 is the lower/higher scale.

Candidate	Step 50	Step 100	Step 150	Step 200
D-1	234 / 1 / 253.8	231 / 0 / 215.3	223 / 0 / 239.9	230 / 0 / 302.5
D-2	182 / 36 / 447.5	176 / 3 / 439.6	190 / 0 / 282.4	128 / 60 / 637.5
L64-1	229 / 0 / 295.0	233 / 0 / 297.8	228 / 0 / 290.8	236 / 0 / 291.7
L64-2	230 / 0 / 209.1	231 / 1 / 205.8	227 / 2 / 223.8	221 / 1 / 236.1
L128-1	234 / 0 / 283.8	234 / 1 / 337.5	229 / 1 / 376.8	232 / 1 / 424.2
L128-2	221 / 4 / 193.2	210 / 18 / 317.3	217 / 14 / 481.0	216 / 4 / 468.9
L256-1	224 / 1 / 286.2	229 / 0 / 291.9	236 / 1 / 298.3	228 / 0 / 259.0
L256-2	224 / 3 / 232.2	84 / 147 / 735.5	225 / 1 / 262.3	209 / 10 / 291.8
S64-1	229 / 0 / 270.1	231 / 0 / 270.9	225 / 0 / 276.1	238 / 0 / 286.0
S64-2	224 / 2 / 307.8	226 / 1 / 271.7	235 / 1 / 285.5	233 / 0 / 309.3
S128-1	232 / 0 / 247.6	227 / 0 / 242.2	224 / 1 / 233.3	224 / 3 / 219.2
S128-2	223 / 2 / 274.4	218 / 1 / 247.5	210 / 4 / 193.4	224 / 0 / 214.3
S256-1	230 / 0 / 282.6	220 / 0 / 228.5	224 / 1 / 231.2	222 / 0 / 228.8
S256-2	187 / 3 / 162.1	0 / 256 / 1024.0	0 / 256 / 1024.0	0 / 256 / 1024.0
N64-1	235 / 0 / 263.8	225 / 0 / 290.7	232 / 0 / 295.9	227 / 0 / 282.7
N64-2	227 / 0 / 240.6	225 / 1 / 288.7	232 / 0 / 287.0	229 / 0 / 284.9
N128-1	227 / 1 / 265.8	229 / 1 / 271.8	228 / 0 / 281.1	234 / 0 / 295.6
N128-2	225 / 0 / 283.2	232 / 1 / 245.2	229 / 3 / 292.4	227 / 2 / 309.5
N256-1	228 / 0 / 324.7	227 / 0 / 334.0	234 / 2 / 348.9	235 / 2 / 343.5
N256-2	230 / 1 / 271.3	227 / 0 / 347.1	230 / 1 / 340.6	228 / 1 / 345.8